University of Zurich[UZH]

A Framework for Rapid and Systematic Software Quality Assessment

A dissertation submitted to the Faculty of Business, Economics and Informatics of the University of Zurich

for the degree of
Doktor der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

by
Martin Alfred Brandtner
from
Lilienfeld, Austria

Accepted on the recommendation of
Prof. Dr. Harald C. Gall
Prof. Dr. Serge Demeyer
Prof. Dr. Martin Glinz

2015

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, October 21, 2015

Head of the Ph.D. committee for informatics:
Prof. Dr. Martin Glinz

Acknowledgements

First and foremost, I want to thank Harald Gall for giving me the opportunity to pursue a fast-track Ph.D. in his research group. I am grateful for his support and guidance, which helped me to handle even difficult times of my Ph.D. studies. No matter how busy his schedule was, he always managed to be an advisor that cares about his Ph.D. students and their research.

Special thanks go to Serge Demeyer and Martin Glinz for being part of my Ph.D. committee as external and additional internal examiner, respectively. I want to thank them both for evaluating my work and for providing valuable feedback. I especially want to thank Serge Demeyer for traveling all the way to Zurich to be part of the committee.

I am grateful to Philipp Leitner, Sebastiano Panichella, Emanuel Giger, Michael Würsch, and Thomas Fritz for the time we spend on research discussions and for their valuable paper feedback. Special thanks to my fellow Ph.D. students Sebastian Müller and Gerald Schermann for their commitment on our successful paper collaborations. Further, I want to thank all fellow Ph.D. students and members of our research group for the pleasant time in the group during my Ph.D. studies.

Finally, I want to thank my girlfriend and my family for their unconditional support during my Ph.D. studies.

Martin Brandtner
Zurich, September 2015

Abstract

Software quality assessment monitors and guides the evolution of a software system based on quality measurements. Continuous Integration (CI) environments can provide measurement data to feed such continuous assessments. However, in modern CI environments, data is scattered across multiple CI tools (e.g., build tool, version control system). Even small quality assessments can become extremely time-consuming, because each stakeholder has to seek for the data she needs. In this thesis, we introduce an approach to enable rapid and systematic software quality assessments. In the context of our work, systematic stands for the ability to do a comprehensive assessment of a software system based on data integrated from different CI tools. Rapid on the other hand, stands for the ability to increase the efficiency during an assessment by tailoring integrated CI data to the information needs of a stakeholder. Based on our findings from mining software repositories and best-practices of practitioners, we present an approach to (i) integrate CI data, (ii) profile stakeholder activities, (iii) tailor the integrated data, and (iv) present it in accordance to the individual needs of a stakeholder. We employed different kinds of studies to evaluate the core concepts of our approach. Additionally, we carried out a user and a case study to investigate the overall potential of our approach in a software quality assessment. The evaluation results clearly indicate that our approach can indeed enable more comprehensive quality assessment within less time.

Zusammenfassung

Das Software Qualitätsassessment überwacht und steuert die Evolution von Software Systemen basierend auf Qualitätskriterien. Continuous Integration (CI) Umgebungen können geeignete Messdaten für solch kontinuierlichen Assessments zur Verfügung stellen. Allerdings sind diese Messdaten in modernen CI Umgebungen über mehrere Anwendungen (z.B. Build-Tool, Versionsverwaltung) verteilt. Selbst kleine Qualitätsassessments können extrem zeitaufwending sein, weil jeder Stakeholder die benötigten Daten individuell abfragen muss. In dieser Dissertation stellen wir einen Ansatz vor, der rasche und systematische Qualitätsassessments von Software Systemen erlaubt. Im Kontext unserer Arbeit verstehen wir unter dem Begriff systematisch die Fähigkeit ein umfassendes Qualitätsassessment basierend auf integrierten Daten von verschiedenen CI Anwendungen zu machen. Der Begriff rasch auf der anderen Seite seht für eine erhöhte Effizienz während Qualitätsassessments, als Resultat von integrierten CI Daten die auf die Informationsbedürfnisse der einzelnen Stakeholder massgeschneidert wurden. Basierend auf unseren Erkenntnissen resultierend von Software Repository Mining and Best Practices von Spezialisten, presentieren wir einen Ansatz zum (i) integrieren von CI Daten, (ii) profilieren von Stakeholder Aktivitäten, (iii) massschneidern der integrierten Daten, und (iv) präsentieren der Daten in Abstimmung mit den individuellen Bedürfnissen eines Stakeholders. Wir setzten verschiedene Arten von Studien für die Evaluierung der Kernkonzepte unseres Ansatzes ein. Die Evaluierungsergebnisse zeigen deutlich, dass unser Ansatz tatsächlich umfassendere und schnellere Qualitätsassessments ermöglicht.

Contents

List of Figures

List of Tables

1

Synopsis

Ever since the beginning of software development, research has investigated different approaches to ensure the quality of software systems. Software testing, software verification, and quality metrics are just a few example areas which have been addressed by research for this purpose. Over the decades, only a handful of approaches reached software development in industry. Starting with the growth of software systems, the importance of software quality increased as well. Crosby argued that *"it is not quality that is expensive, but rather the lack of it"* [Crosby, 1979]. It is therefore not surprising that some estimates (e.g., [Brooks, 1995]) claim that around 80% of information system budgets are spent on the maintenance of a software system.

An effective and widely followed approach to detect potential quality flaws in source code is the use of Continuous Integration (CI). CI is an approach to automatically integrate, build, test, and determine quality metrics (e.g., complexity) of source code contributions. Today, multiple tools in a CI environment form a tool chain in which every link works together with the others.

There are approaches which provide a tighter integration (e.g., Rational Team Concert), but those are applicable for new software projects only. A migration of existing software projects into such an integrated environment is very complex or even impossible. In this thesis, we are focusing on software projects that do not use such a tightly integrated CI environment.

Besides an automation of the source code integration it is important to regularly assess the quality of a software system to detect quality flaws other than, for example, build or test failures. The integration of data originating from different CI tools is a crucial requirement to enable such regular assessments of the overall quality of a software system. In software quality assessments, stakeholders describe the quality status of a software system from different perspectives reaching from code quality to the quality of the release and issue management. In the context of this thesis, the term stakeholder stands for the members of a software project in general and software developers in particular. The individual stakeholders of a software quality assessment gather the required information from dashboards that are offered by a CI environment. Those dashboards provide an overview of the data that is available in a certain CI tool. Due to the data scattering across multiple tools and a generic data presentation, each stakeholder has to individually query data in order to draw a conclusion about the different quality aspects of a software system. It is therefore important to reduce the required effort to access the according data for each individual stakeholder.

1.1 Motivation

The aim of a software quality assessment (SQA) in the context of this work is the monitoring and guidance of a software system's evolution based on quality measurements (e.g., code metrics) and the project status (e.g., build/release status). There are various ways and aspects that can be followed or addressed to achieve this goal. Traditional research in the field of software quality is focusing primarily onto single aspects to describe the quality of a software system based on, for example, quality metrics (e.g., [Lanza et al., 2005]) and test

coverage (e.g., [Mockus et al., 2009]). However, further research (e.g., [Mockus et al., 2002, Zimmermann et al., 2007, Wu et al., 2011]) has shown that the utilization and interlinking of version control system (VCS) data and issue tracker data can foster the understanding of software maintenance. Those findings highlight the importance of assessing the quality of a software system as a whole instead of focusing on selected technical measurements, such as code metrics only. In this thesis, we put a special focus on the compliance of software projects with respect to well-established best practices in source code integration.

We claim that ideally SQA is accomplished during the life-cycle of a software system, starting with the first source code file and ending with the fade out of production. Today, modern CI environments cover almost the complete life-cycle of a system. CI is an important element to achieve a certain level of software quality. The data available in modern CI environments covers build related data (e.g., build failure), testing data (e.g., test failures), quality measurements (e.g., metrics), and release management data (e.g., commit/issue management).

However, a regular assessment of all the data available in CI environments is time-consuming and depending on the size of a software system even impossible. The large number of covered aspects per integration run and the short integration cycles (multiple integration runs per day) raise the need for a mechanism to tailor the multitude of CI data. A further challenge is the scattering of the data within a CI environment. Modern CI tools provide dashboards to visualize the data generated during an integration run. In such dashboards, stakeholders have to individually seek the needed information. This is an extremely time-consuming task, because each tool presents its own data only. The scattering raises the need for an integration of the data generated by different tools. Additionally, the integrated and tailored CI data has to meet the information needs of each individual stakeholder to gain efficiency during SQA. Information needs of different stakeholders in a software project were investigated in related work (e.g., [LaToza and Myers, 2010, Fritz et al., 2010]).

Based on these findings, it is possible to compose individual views for different stakeholders.

However, the employment of agile software development processes makes it hard to assign a single role to a stakeholder. For example, a software engineer has to implement and test a source code change. In such a case, a software engineer has two roles, developer and tester. The information needs of a stakeholder therefore depend on her activity within the time span of a quality assessment. Modern CI tools do not explicitly store activity data, but each change in a CI tool is tracked with a time stamp and the user name of the performing stakeholder. These data can be utilized to construct activity logs for every stakeholder. We want to use the activity log data in combination with best-practices to enable the composition of integrated and tailored views for individual stakeholders. Every best-practice describes a guideline and lists the artifacts, which have to be checked in order to ensure the compliance of a software project. The list of artifacts can be matched with the activity logs of the different stakeholders. Based on this artifact matching, it is possible to estimate the importance of certain CI data for an individual stakeholder.

Thesis Statement. In software quality assessment, stakeholders will benefit from integrated and tailored views on software quality data generated out of CI environments to enable more comprehensive and more efficient quality analysis.

1.2 Context and Scope

According to McConnell [McConnell, 2004], quality of software can be categorized into two groups, internal quality and external quality. The research focus of this thesis is onto the internal quality of a software system in general, and onto the development process quality in particular.

Recent research [Wu et al., 2011] has shown that especially the meta data (e.g., commit message, comments) stored in VCS and issue trackers can be

systematically mined and analyzed for quality assessment purposes. The investigations presented in this thesis primarily address data from continuous integration, VCS, and issue trackers. We mined and analyzed the respective data from open-source Java projects hosted by, for example, the Apache Software Foundation, because those projects are actively maintained and exist for more than five years. We further decided to focus our research onto developers and testers, because those roles are defined within the analyzed open-source projects and they interact with the according tools. However, it is possible to extend and adopt the presented work for other stakeholders and quality measurements.

1.3 Research Questions

The overall goal of this thesis is to support different stakeholders in software quality assessment. We aim for an integration and tailoring of CI data in accordance with the information needs of stakeholders. With respect to the overall goal and our thesis statement, we define the following three research questions. Figure 1.1 depicts an overview of the research questions and their relationship.

Research Question 1 (RQ1): *How can data of current VCS and issue trackers be utilized for quality assessment right before a software release?*

The first research question investigates best-practices for software quality assessment and the use of data originating from VCS or issue trackers. We are especially interested in software quality assessments that take place right before a software release. At such a point in a software project, a large amount of source code changes have to be tested and integrated into the according branches of a codebase (e.g., back porting of security fixes). Interviews of practitioners should help us to understand the most important quality checks and information sources that are needed to effectively execute those checks. We will document the findings of

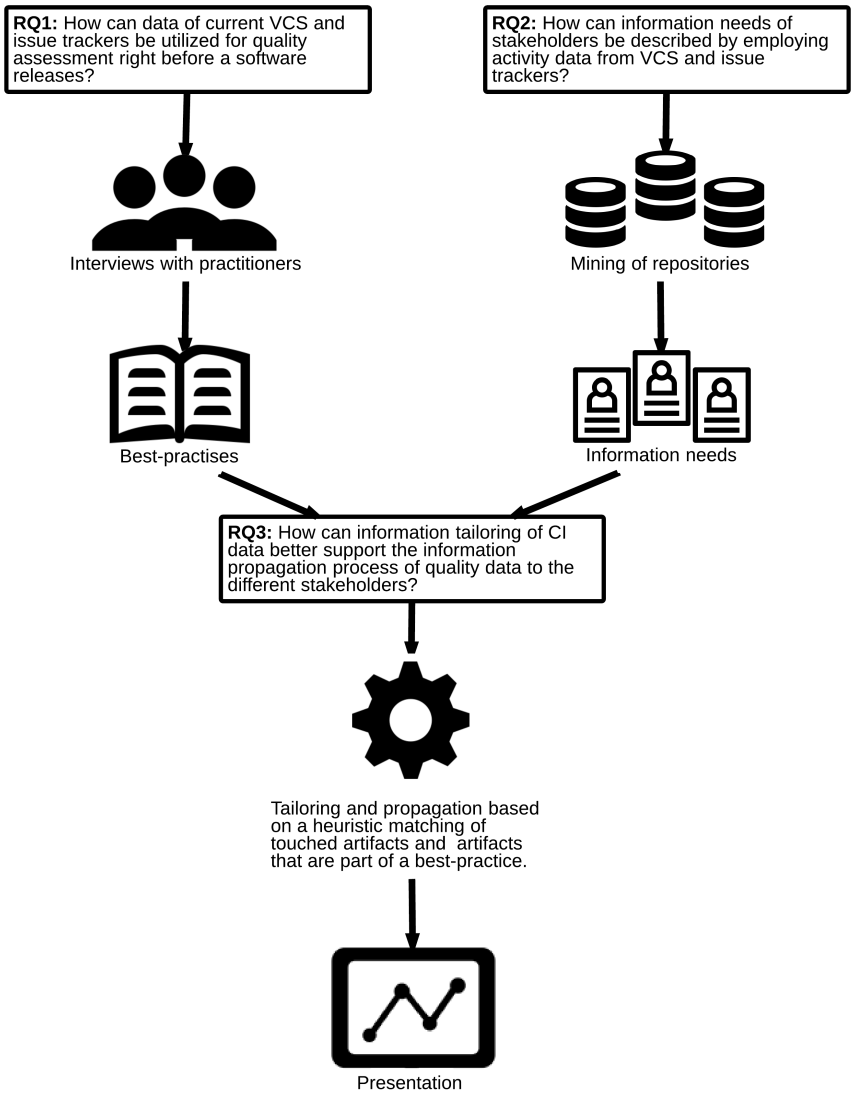

Figure 1.1: Relationship between the Research Questions

this research question with a set of pragmatic recipes, which represent generalized and project-independent best-practices.

In a mining study, we will investigate the degree to which modern open-source projects follow the proposed recipes. Based on the findings of earlier research [Bachmann et al., 2010, Wu et al., 2011], we expect that those recipes will be followed in practice. As with heuristics, we do

expect deviations as our recipes do not constitute formal rules. Every deviation can negatively influence the efficiency of a recipe. In order to overcome recipe deviations, we will propose an automatic approach to optimize the structure in VCS and issue trackers for the use with pragmatic recipes.

Research Question 2 (RQ2): *How can information needs of stakeholders be described by employing activity data from VCS and issue trackers?*

The second research question addresses the description of information needs based on activity data. So far research on information needs in software engineering (e.g., [LaToza and Myers, 2010, Fritz et al., 2010]) has focused on the role of a stakeholder. The role, on the other hand, describes the activities of a stakeholder. We envision an approach to determine the information needs of a stakeholder based on its activities. In software development, stakeholder activities are related to activities in tools or repositories. For example, a stakeholder commits a code change to the VCS. This stakeholder activity is reflected as commit entry in the VCS.

We will investigate VCS and issue tracker data for potential approaches to enable a mapping of tool activities to stakeholder activities. Additionally, we will investigate stakeholder activity data with a focus on activity patterns. Potentially existing activity patterns can be used for a mapping of stakeholder activities to stakeholder information needs.

Research Question 3 (RQ3): *How can information tailoring of CI data better support the information propagation process of quality data to the different stakeholders?*

The final research question addresses the information propagation process of quality data originating from independent tools to the stakeholders of a software quality assessment. We are especially interested in the integration of data from different CI tools and the data tailoring based on information needs of stakeholders. As starting point of our research,

we will use a literature survey with a focus on development tool design. Based on the survey outcome, we will sketch and implement a prototypical framework for quality data integration and tailoring. The resulting framework can be used for initial investigations on the strengths and weaknesses of the envisioned integration and tailoring approach.

We carried out a user study with two groups of software engineers. Both groups have to solve the same program comprehension and software testing related tasks, but with different tools. One group has to use modern CI tools and the other group has to use our prototypical framework. Based on the study results, we will conclude on the efficiency gains and the usability of our prototypical framework.

1.4 SQA-Cockpit in a Nutshell

In this thesis, we devised a framework that supports individual stakeholders with tailored data to enable more comprehensive and faster software quality assessment. The major aim of this approach is to enable full-fledged software quality assessments with a minimum of information gathering effort.

Our approach is called *SQA-Cockpit*. This name is inspired by modern cockpits of vehicles, such as aircrafts. Those cockpits integrate data from different systems (e.g., engine, landing gears) and tailor these data to the needs of the pilot. SQA-Cockpit integrates data from different tools (e.g., build tools, VCS, issue tracker) and tailors these data to the needs of stakeholders in a software quality assessment. In both scenarios, the primary aim is to provide data that is required to keep an overview of the situation, without overwhelming the person in charge. Figure 1.2 depicts an automatically composed view of the SQA-Cockpit with data originating from four different CI tools. The presented data is already tailored based on the activity history of the according stakeholder.

Furthermore, SQA-Cockpit provides capabilities to tailor data for a certain time span. For example, the view depicted in Figure 1.2 contains only data that was generated or changed within the last two weeks. The *Code compliance*

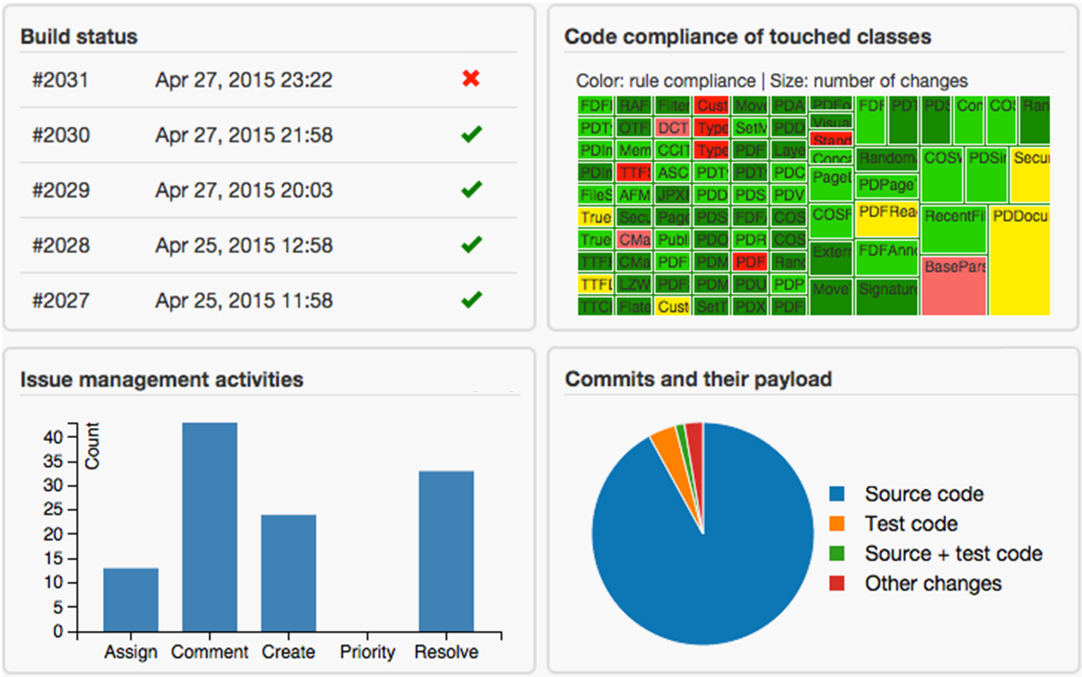

Figure 1.2: SQA-Cockpit – Layout of an Automatically Composed View

of touched classes widget on the right top in this figure, presents a treemap widget with source code changes of the given time span. Each rectangle in this treemap widget represents a source code file that was changed by the viewing stakeholder. The size of a rectangle reflects the number of changed lines of code within a file. The color indicates the evolution of the code quality, which is a direct result of the underlying source code changes. Another characteristic of SQA-Cockpit is the ability to dig-deeper into the presented data. For example, it is possible to click on elements of widgets to get more information about it. In case of the *Issue management activities* widget on the left bottom in Figure 1.2, a click on a bar opens a new window that presents a list of effected issues and an event log of the according activity. When the presented data is still not sufficient to satisfy the information needs of a stakeholder, a list of links is available to directly open the according entries in the issue tracker.

SQA-Cockpit is not limited to the presented kind of widgets or data sources. It currently supports the handling of data originating from build tools (e.g.,

Jenkins-CI), issue tracker (e.g., JIRA), quality management platforms (e.g., SonarQube), and VCS (e.g., Git). The handling of data origination from other kind of CI tools is possible, due to the clear separation of concerns followed during the design of SQA-Cockpit.

In the remaining chapters of this thesis, we show additional use cases in which SQA-Cockpit can foster software quality assessment. We further provide a detailed description of the underlying data integration and tailoring approach as well as the used profile extraction and presentation approach.

1.5 Foundation of the Thesis

The foundation of this thesis is a collection of selected publications that were published in international, peer-reviewed venues in the area of Repository Mining and Software Maintenance.

Chapter 2: *Intent, Tests, and Release Dependencies: Pragmatic Recipes for Source Code Integration*

Goal. Identification of pragmatic recipes for supporting the decision making during release management based on data stored in repositories.

Contribution. In this work, we report on an interview study that asked practitioners from industry to capture their best-practices for release management. Our interview partners were experienced senior developers, lead developers, and technical software architects from four software companies located in Switzerland. Based on the findings of the interview study, we derived an initial set of three pragmatic recipes plus variations. Recipes are built on project independent best-practices, and consist out of a rule (the actual guideline), one or more ingredients (artifacts or data), optional variations (for project-specific requirements), and a purpose (why a recipe is useful). The rules of the pragmatic recipes can be summarized as follows:

- **Reveal the Intent.** Contributions should contain at least one link to an issue.

- **Align the Tests.** Changes in source and test code should be combined in one commit.
- **Reveal the Release Dependencies.** Commits that address multiple issues should be marked as such.

In a next step, we mined VCS and issue tracker data of 21 open-source projects. The mined data were used to investigate cases whether or not recipes are followed. We performed a quantitative analysis to see to what extent the proposed recipes can be identified in open-source projects. Additionally, a qualitative analysis was applied to identify potential reasons for recipe deviations. The results of the quantitative analysis showed that two-thirds of the analyzed open-source projects follow at least two of the proposed recipes in most of the cases ($> 75\%$). In the remaining cases, recipe deviations were mostly caused by small source code changes (e.g., fixing typos), which are exceptional cases mentioned by our interview partners.

Conclusion. With this work, we were able to show that software development follows recipes, which are neither explicitly defined nor enforced within development tools. The use of such recipes is reflected in the data stored in VCS and issue tracker. The ability to mine data of current CI environments for software quality assessment purposes, such as release management is a huge benefit compared to switching to a fully-integrated solution, such as Rational Team Concert. For example, a fully-integrated solution can hardly facilitate data that was generated before its installation. We successfully showed that pragmatic recipes make it possible to systematically analyze the evolution of a software system, without installing a fully-integrated development environment.

Research Question. With this work, we address ways to utilize data of current CI environments for software quality assessment (**RQ1**).

Chapter 3: *Discovering Loners and Phantoms in Commit and Issue Data*

Goal. Automatic detection of commits without issue reference and ap-

proaches to link those commits with related issue entries to make them visible and usable for software quality assessment.

Contribution. In this work, we investigated the linking of artifacts in modern VCS and issue tracker. We introduced a formal model called *PaLiMod* to simplify the residual handling. In the context of this work, a residual of commits incorporates all commits that have no issue key in the commit message. The *PaLiMod* model builds on Semantic Web technologies and has a strong focus on the linking of commit and issue data. We used the model for our analysis of the commit and issue data of 15 Apache projects. A first analysis of the data showed that the mean residual of non-linked artifacts is below 26%. In a second step, we extracted the characteristics of non-linked artifacts. We were able to determine two scenarios where a link is not explicitly established and therefore not visible:

- **Loner.** This characteristic reflects situations in which one commit is associated to one issue and vice-versa, but the link is missing.
- **Phantom.** This characteristic reflects situations in which an issue has multiple associated linked commits, but there are further commits that are associated with, but not linked to the issue.

In a further step, we derived heuristics to automatically detect and establish links for *Loner* and *Phantom* scenarios. Both heuristics use person, time, and resource data to detect and link artifacts within the described scenarios. The evaluation of our approach showed that both heuristics achieve good results to further reduce the residual of non-linked artifacts. Overall, the *Loner* heuristic achieves a precision of 96% and a recall of 92%. The *Phantom* heuristic achieves a precision of 73% and a recall of 53%.

The author of this thesis acted as second author in the original work [Schermann et al., 2015] presented in this chapter. The original work was inspired by the outcome of the investigations on recipes, which were

introduced in the previous section. The author of this thesis used the results of the recipe investigations to motivate and propose a model-based link establishing approach for VCS and issue tracker data.

Conclusion. With this work, we were able to show that artifact linking is heavily used in software projects. Software quality assessment can benefit from artifact linking, because related artifacts can be effectively accessed without needing long lasting searches. However, in such a scenario, it is extremely important to consider the re-establishment of links between artifacts, to increase their visibility in dashboards of modern development tools. A stakeholder has to be aware of this fact and to actively search for non-linked artifacts to get information about them. The *Loner* and *Phantom* heuristics are just two examples, which increase the level of artifact linking to foster software quality assessment.

Research Question. In this work, we address the impact of commits that are not linked to other artifacts with respect to pragmatic source code integration recipes (**RQ1**).

Chapter 4: *SQA-Profiles: Rule-Based Activity Profiles for CI Environments*

Goal. Identification of information needs based on stakeholder activities in the version control system and issue tracker.

Contribution. In this work, we investigated VCS and issue tracker data of 20 open-source projects to mine activity profiles of stakeholders. Our approach was to extract activity data from event histories, because modern VCS and issue tracker do not support an explicit tracking of stakeholder activities. The artifact-centric manner of event histories (e.g., each issue has its own history) in modern VCS and issue tracker required a mapping of event history entries to gain the activity data of stakeholders. For each stakeholder of a software project, we mined events related to assignee, comment, commit, merge, priority and status changes. In the following, we refer these events as attributes. A clustering of the activities per stakeholder resulted in four distinctive profiles:

- **Bandleader.** This profile describes a stakeholder that has a high activity in each attribute. A stakeholder with this profile keeps the music playing in a project, and it is very likely that the music stops when such a stakeholder leaves the project.
- **Integrator.** This profile describes a stakeholder that has a high merge activity in the VCS, and at least one other attribute with moderate activity. A stakeholder with this profile primarily handles the integration of source code contributions in a software project.
- **Gatekeeper.** This profile describes a stakeholder that has high activity in status changes and a moderate activity in assignee changes or commits. A stakeholder with this profile decides when the status of an issue gets changed.
- **Onlooker.** This profile describes a stakeholder that only occasionally contributes to the VCS and the issue tracker of a software project.

We further proposed a ruled-based model to enable a project-independent description of activity profiles. For this, we introduced a nominal scale with the values *Low*, *Medium*, and *High*, which describes a stakeholder's activity within an attribute. For example, a *High* commit activity. It is important to mention that each scale is computed per project, which means the absolute value, for example, for *High* varies depending on the project. In the evaluation, we compared the performance of our automatic ruled-based approach against a semi-automatic analysis with machine learning. The results showed that our rule-based and project-independent approach can be used to automatically extract the profiles of stakeholders with a precision of 92% and a recall of 78% compared to the dataset extracted by a project-dependent and semi-automatic approach.

Conclusion. With this work, we were able to show that ruled-based profiles are a reliable approach to describe information needs of stakeholders based on data extracted from VCS and issue tracker. Profiles based on activity data can be used to flexibly describe the role of stake-

holder. This is especially important for software projects that follow an agile development process. In agile software development, the role of a stakeholder can change depending on the phase of a sprint. For example, a software engineer implements a new functionality and in a later phase, she takes care of the same functionality. During the implementation, a stakeholder is interested in different information than during the testing. Stakeholder profiles are a promising approach to handle the information need determination in a rapidly changing environment.

Research Question. In this work, we address the information need extraction from activity data (**RQ2**).

Chapter 5: *SQA-Mashup: A Mashup Framework for Continuous Integration*

Goal. Identification of an information tailoring approach to increase the efficiency of software quality assessment.

Contribution. In this work, we investigated approaches to tailor information in accordance with the needs of different stakeholder roles in software quality assessment. We used a literature survey to determine basic tailoring requirements for different roles in software development. As result of the survey, we decided to apply a mashup-based paradigm for data integration and tailoring. The mashup-based paradigm enables a flexible way to select, filter, and merge data extracted from different data sources, such as CI tools. Our proposed data integration and tailoring approach is built on-top of these basic operations, which can be combined to more complex operations. With the help of a prototypical mashup engine, we setup a quality data integration and tailoring scenario for software developers (data related to program comprehension) and testers (data related to testing). We used views consisting of widgets to visualize the integrated and tailored data. A widget in the context of this work can be, for example, a list, a chart, or a treemap. We carried out a user study to estimate the potential of our approach and its impact on the efficiency of a software quality assessment. In a second literature survey, we looked out for information needs of stakeholders that cover

program comprehension, testing, and general questions about a software system. This survey results in nine independent tasks (three for each topic), which we asked to solve the participants of our user study. In two groups, study participants had to solve as many tasks as possible. The maximum time limit per task was five minutes. The control group had to use modern CI tools (GitHub, Jenkins-CI, SonarQube) and the experimental group had to use the widget-based views for developers and testers. In both cases, we used only data from the mentioned CI tools. The results of the user study showed that the participants of the experimental group solved the tasks of our study faster (57%) and with a higher correctness (21.6%). When analyzing these differences on the level of the individual tasks we found evidence that monitoring software quality during CI can substantially benefit from integrated and tailored data. This is particularly the case when the required information is scattered over multiple CI-tools and displayed in different scales and units.

Conclusion. With this work, we were able to show that it is possible to integrate and tailor data from different CI tools for enabling a fast and accurate propagation of quality information. Both groups in the user study had exactly the same information available, but the study results indicate clear weaknesses of how information can be accessed in modern CI tools. With a pragmatic separation of information based on stakeholder roles, we were able to compose views that help stakeholders in getting the needed information. The results showed a great potential for tailored views in combination with profiles instead of roles. The use of profiles enables a more fine grained tailoring compared to roles and the views evolve with the actual activity of a stakeholder.

Research Question. In this work, we address the information propagation and tailoring (**RQ3**).

Chapter 6: *A Framework For Rapid and Systematic Software Quality Assessment*

Goal. Introduction of a framework that enables better quality assessment

of a software system by integrating and tailoring of data in accordance with information needs of stakeholders.

Contribution. In this work, we introduced a formal model for data integration, tailoring and presentation based on stakeholder profiles to enable rapid and systematic software quality assessments. In the context of this work, systematic stands for the ability to do a comprehensive quality assessment based on data from different CI tools, and rapid stands for an increase efficiency by a reduced information seeking time. We additionally introduced a CI data processing framework called SQA-Cockpit on top of the formal model. This CI data processing framework implements following three phases:

- **Integration.** This phase covers the extraction and linking of artifacts (e.g., builds, commits, issues).
- **Tailoring.** This phase covers the stakeholder profiling as well as the filtering and preparation of integrated data for the presentation to a certain stakeholder.
- **Presentation.** This phase covers the visualization of the integrated and tailored data.

The data processing concept of SQA-Cockpit enables a fully-automatic profiling, tailoring, and view composition based on the user name of a stakeholder. All data required for the calculations can be mined from the different CI tools. For the implementation of the calculations, we rely on the findings of the work presented in Chapters 2-5. Currently, SQA-Cockpit supports GitHub, Jenkins-CI, JIRA, and SonarQube as source systems, but it is possible to extend the range of supported CI tools by implementing an adequate connector.

Conclusion. With this work, we were able to show that supporting stakeholders with tailored information can enable a more comprehensive and faster software quality assessment. The only data source needed for such a gain of efficiency are exactly the same tools that are already

used for software quality assessment. We applied repository mining techniques to gain information which is invisible or not processable by a human. This information we use to prioritize the data that is visible and accessible for a stakeholder during a quality assessment. The result is a better utilization of already existing data stored in CI tools, which can positively influence the quality of a software system.

Research Question. With this work, we address the overall research goal of our work by combining the findings of the single research questions into a ready-to-use framework.

1.6 Summary of Contributions

Regular software quality assessment based on data available in CI environments is time-consuming and depending on the size of a software system even impossible. The large number of covered aspects per integration run and short integration cycles raise the need for a mechanism to automatically tailor data to the information needs of a stakeholder.

In this thesis, we investigated approaches to integrate, tailor, and present CI data to enable rapid and systematic software quality assessments. The major contributions of this thesis are as follows:

Pragmatic Recipes. We provided a set of pragmatic recipes based on best-practices in assessing quality of a software system before its release. These recipes describe, for example, guidelines to link every commit to an issue in order to foster the information gathering process during a software quality assessment. Our investigations on pragmatic recipes showed that software companies as well as community projects (e.g., Apache Software Foundation) follow these recipes to a large extent.

Deviation handling in Recipes. We introduced a heuristic approach to automatically detect and resolve the residual of cases, which deviate from the proposed recipes. For example, our heuristic is able to link commits

and issues in cases where the commit message does not contain a unique identifier (e.g., issue-key).

Stakeholder Profiling. We introduced a profiling approach based on stakeholder activity data extracted from VCS and issue tracker to enable an automatic tailoring of quality data for software quality assessment. This approach enables a categorization of different stakeholders based on their activities in development tools. Profiles are more accurate compared to stakeholder roles, because profiles rely on the actual activities. This is especially important for agile development where engineers have a sightly different work focus depending on the phase (e.g., a developer implements test code).

Quality Mashup. We introduced a mashup-based approach to tackle the integration and tailoring of data structures originating from different CI tools. The mashup concept enables a simple and flexible way to select, filter, and merge different data structures. Despite its simplicity, it is possible to combine the single mashup operations to more complex integration and tailoring scenarios. The mashup itself is not visible to a user of the tool. Instead, a stakeholder sees the resulting data visualized in a dynamically composed view consisting out of various type of charts.

Framework. We developed two proof-of-concept frameworks, which both aim for CI data integration and tailoring. The frameworks reflect the evolution of our approach. We started with the SQA-Mashup, which provides role-based views and tailoring of integrated CI data. The SQA-Cockpit framework, on the other side provides profile-based views in combination with profile and time-based data tailoring. For example, SQA-Mashup always presents the whole evolution of a software system from the perspective of a certain role. SQA-Cockpit allows for presenting the evolution of artifacts touched by a certain stakeholder and within a predefined time span.

We employed empirical and user studies to evaluate the core concepts of

our approach. Additionally, we carried out a user and a case study to investigate the overall potential of our approach in a software quality assessment. The evaluation results clearly indicate that SQA-Cockpit can indeed enable faster and more comprehensive quality assessment.

1.7 Limitations and Future Work

We have demonstrated that our CI data integration and tailoring approach can enable more comprehensive and faster software quality assessment. However, evolving capabilities of tools in CI environments lead to limitations and challenges that have to be addressed in future work. In the following, we present a list of limitations and challenges for future work.

Stakeholder Profiles. We derived a set of stakeholder profiles based on our proposed profiling approach. For the establishment of these profiles, we extracted a specific set of data attributes from VCS and issue tracker. We ensured that there are no correlations between the selected attributes. However, further investigations are required to investigate whether additional attributes do exist that are suitable for stakeholder profiling. Another limitation of our profiling approach is the strong focus on activity data extracted from VCS and issue tracker. At the current stage, our approach assumes that software quality assessments are done by stakeholders that actively contribute to the VCS or issue tracker. However, these assumptions do not hold in general. In some projects, stakeholders (e.g., software architect) work with additional repositories (e.g., document repository) and have only a few or no activity in the VCS or issue tracker [Aranda and Venolia, 2009]. The extraction and use of activity data from additional tools can reduce this limitation.

Information Needs. In research, it is a common practice to describe information needs of stakeholders with respect to their roles. However, in Chapter 4 we were able to show that stakeholders within the same role (e.g., project management committee member) can be further grouped

into fine-grained profiles. This finding directly impacts our work as we profile stakeholders based on their activities, but at the same time, we rely on findings from information needs research for the information tailoring. Our approach is able to derive the fine-grained profile of a stakeholder, while the information tailoring is limited to the role level. Future work in the field of information needs with a focus on profiles can help to overcome this limitation.

Pragmatic Recipes. In Chapter 2, we proposed a set of pragmatic recipes for release and quality management based on best-practices from practitioners. We were able to show that practitioners from different companies use similar approaches and look at similar quality measurements. For example, the use of issue-keys in commit messages is highly appreciated in order to allow for a seamless tracking of issues and source code changes. Despite these similarities, it is hard to establish project-independent recipes because of a few exceptional cases which exist in each of the companies. We tried to address this limitation with recipe *variations*, which basically adopt a recipe by including project-specific requirements. In a mining study of Apache projects, we were able to show that our recipes with variations exist in open-source projects as well. Many projects contain variations of recipes that co-exist concurrently. However, our recipe concept is limited to follow only one recipe variation within a software project. To overcome this limitation, it would be necessary to rethink the way of handling exceptional cases in recipes.

Tool Integration. The integration of data from different CI tools was a central claim of our work. Also, more and more modern CI tools begin to offer integration and linking of data from other CI tools. For example, JIRA extracts issues-keys from commit messages to enable a linking of issues to commits. However, neither our approach nor the JIRA approach supports a propagation of calculated artifact links (e.g., commits to issues). For example, a stakeholder can benefit from data integration as long as she uses JIRA or SQA-Cockpit. On the other side, the same stakeholder

cannot benefit from those links when using the VCS (e.g., GitHub) or any other CI tool. Future work should investigate approaches for link propagation to enable bi-directional linkings.

Visualization. Our approach aims for a fully-automated view composition based on the profile of a stakeholder. Even if the envisioned approach works perfectly fine, it can be necessary to manually query and visualize data in exceptional cases. Currently, our approach does not provide a query language for manual queries. Future work should investigate the need for such individual queries. It could well be that the implementation of a simple query language is enough to address this potential limitation.

1.8 Roadmap of the Thesis

The remainder of this thesis consists out of five chapters, which form the overall contribution of our work. Figure 1.3 depicts a roadmap of the thesis based on the single chapters and the research questions. Each chapter is based on a publication as depicted in Figure 1.4.

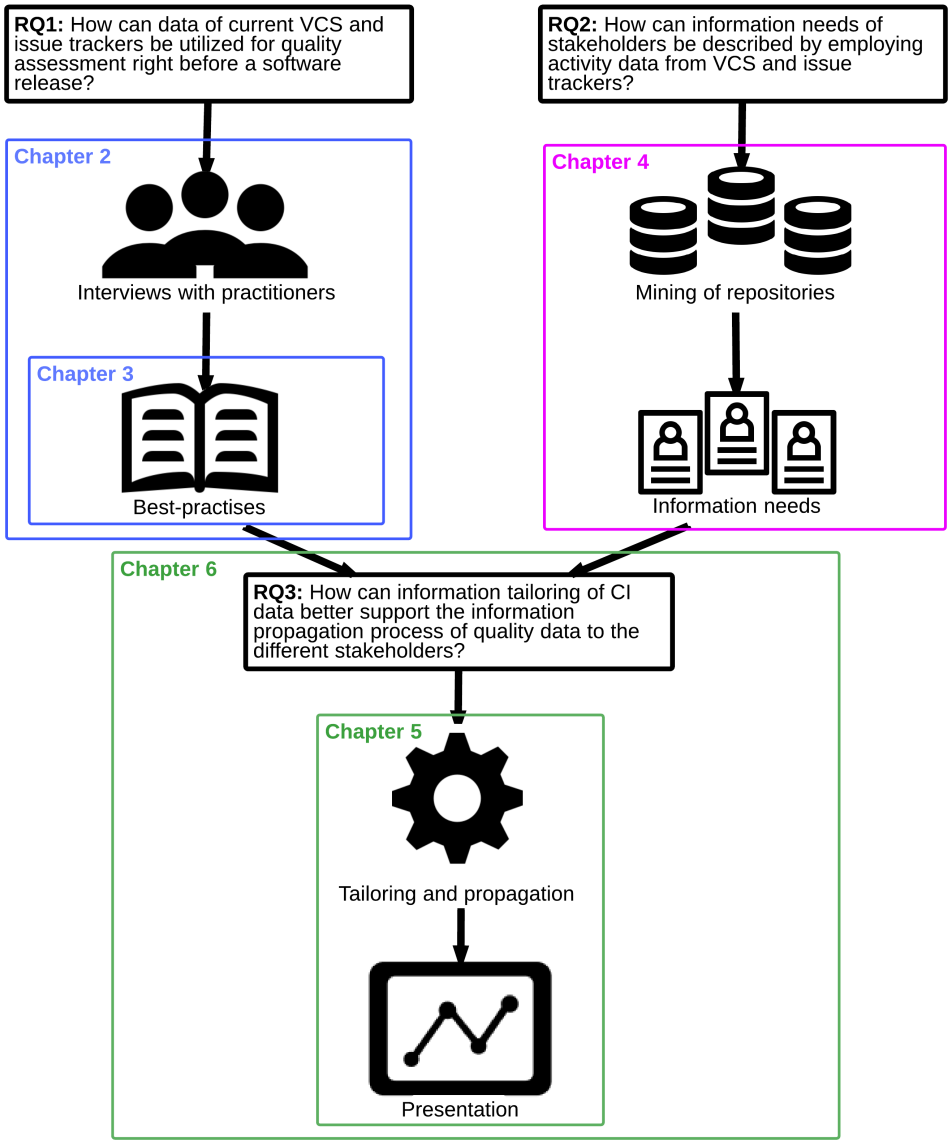

Figure 1.3: Thesis Roadmap

The publications are sorted in the intended order of reading.

Chapter 2 provides an overview of best-practices in assessing the quality of a software system right before its release. A special focus of this chapter is on the utilization of VCS and issue tracker data for software quality assessment.

Chapter 3 presents a set of heuristics to automatically establish links between artifacts from VCS and issue tracker. The results of Chapter 2 showed that artifact linking is an important element of best-practices, which can foster the assessment of software quality.

Chapter 4 introduces an automatic profiling approach for stakeholders of a software project. This approach works with activity data extracted from development tools and builds an important element for the envisioned information tailoring.

Chapter 5 introduces a mashup-based approach for CI data integration and tailoring based on stakeholder roles. The user study presented in this chapter shows the potential of the mashup-based approach for enabling faster and more accurate software quality assessments.

Chapter 6 presents a cumulative version of our approach, which was followed in this thesis. In this chapter, we present a refined and final version of our mashup-based approach.

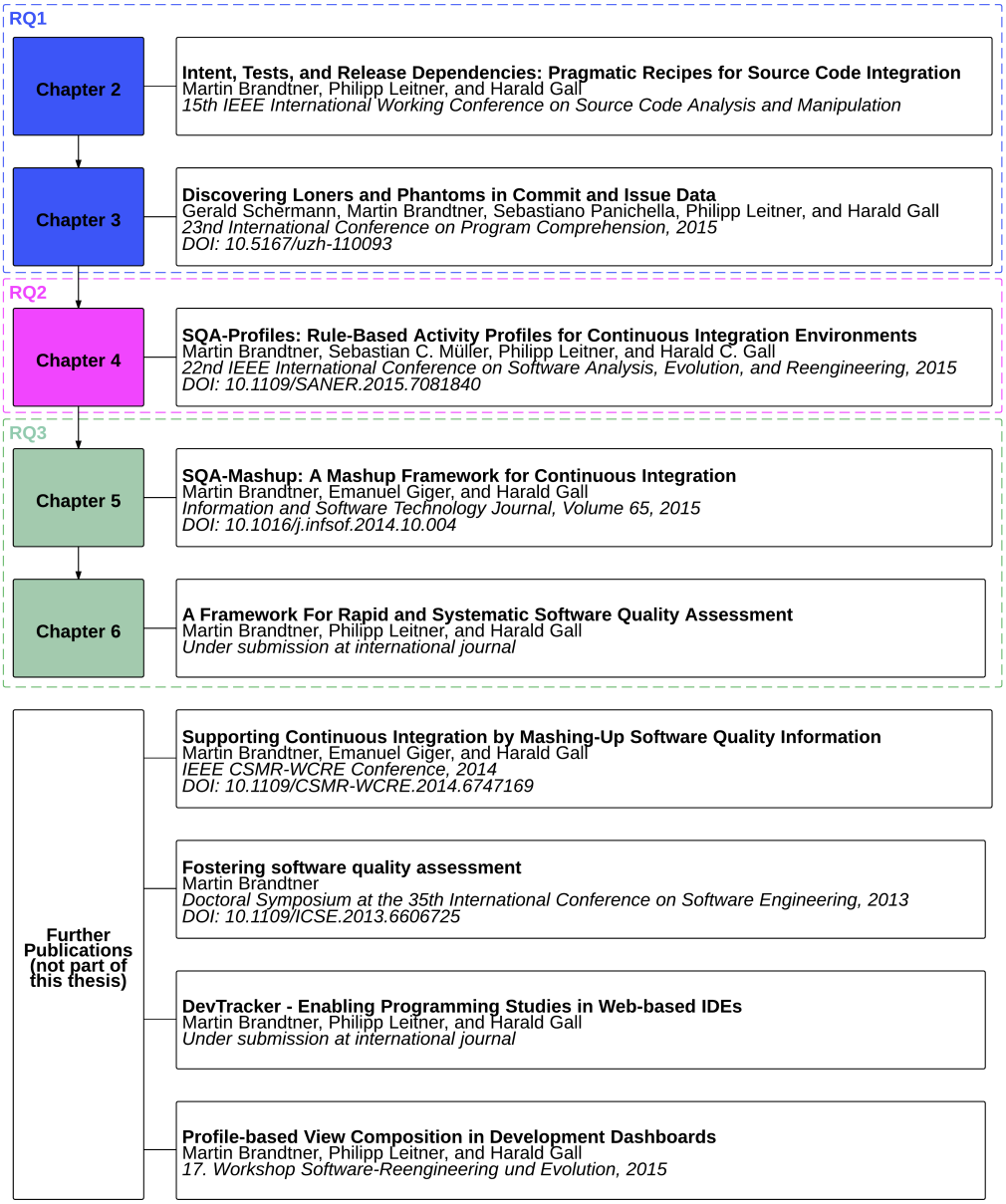

Figure 1.4: Thesis Chapters and Scientific Publications

2

Intent, Tests, and Release Dependencies: Pragmatic Recipes for Source Code Integration

Martin Brandtner, *Philipp Leitner, and Harald Gall*
Published at the 15th IEEE International Working Conference on Source Code Analysis and Manipulation, 2015

Abstract

Continuous integration of source code changes, for example, via pull-request driven contribution channels, has become standard in many software projects. However, the decision to integrate source code changes into a release is com-

plex and has to be taken by a software manager. In this work, we identify a set of three pragmatic recipes plus variations to support the decision making of integrating code contributions into a release. These recipes cover the isolation of source code changes, contribution of test code, and the linking of commits to issues. We analyze the development history of 21 open-source software projects, to evaluate whether, and to what extent, those recipes are followed in open-source projects. The results of our analysis showed that open-source projects largely follow recipes on a compliance level of $> 75\%$. Hence, we conclude that the identified recipes plus variations can be seen as wide-spread relevant best-practices for source code integration.

2.1 Introduction

Over the last years, the importance of continuously integrating source code contributions has raised in software development. Pull-request-driven contribution channels (e.g., GitHub [Gousios et al., 2014, Tsay et al., 2014]) open comfortable ways to submit contributions to software projects. Continuous integration encompasses various kinds of platforms, including (1) version control systems (e.g., Git), (2) issue tracking (e.g., JIRA), or (3) build and release management systems (e.g., Jenkins). These platforms, as well as the artifacts associated with them, are not independent.

In this paper, we focus on the intersection of source code contributions and issues during the integration process. Generally, there are two different classes of problems that software managers face related to this intersection. Firstly, **feature and release management** requires software managers to stay aware of which feature or bug fix is contained in which contribution. Further, release management also requires isolation of changes, so that, for example, single feature implementations can easily be integrated into specific releases. Secondly, **contribution quality management** requires software managers to make sure that source code contributions submitted for issues are of sufficient quality and actually implement the required change.

In practice, various conventions and guidelines have been established to

support software managers with feature, release, and quality management. For example, to ease release management, the Apache Software Foundation (ASF) has established a guideline [Apache Software Foundation,] that every commit message has to contain a reference to an entry in the issue tracker via the issue identifier. Similar guidelines exist in a plethora of other software projects and can help to avoid non optimal source code changes (e.g., [Zimmermann et al., 2012, An et al., 2014]). However, those guidelines are generally neither well-defined, nor portable between projects, nor are there tools to support their convenient monitoring.

In this paper, we propose a set of pragmatic guidelines, referred to as **recipes**, to foster the interlinking of software development artifacts and tools. Recipes are built on project-independent best-practices, and consist of a **rule** (the actual guideline), one or more **ingredients** (artifacts or data), optional **variations** (for project-specific requirements), and a **purpose** (why a recipe is useful). We interviewed software managers to capture their best-practices for the definition of an initial set of recipes. A software manager in the context of our work has one of the following roles: lead developer, senior developer, or senior software architect. In the second step, we analyzed the development history of 21 open-source projects from the ASF, JBoss, and Spring to see to what extent the proposed recipes can be identified in open-source projects. A manual monitoring of even simple recipes is extremely time-consuming. For example, the history of the Apache Ambari project contains 2577 commits from 39 different contributors between July 2014 and December 2014. In the same time period, 2450 issues were created or resolved. It is obvious that it is not feasible for a software manager to manually keep track of every single source code contribution or issue and its impact, especially if she is involved in multiple projects at the same time. Hence, we have devised semi-automated tooling to foster the inspection of existing projects for compliance with the proposed recipes.

The three main contributions of this paper are:

- The definition of **three pragmatic recipes plus variations** for source code integration based on best-practices in three software companies.

- A **quantitative analysis** of 21 projects from ASF, JBoss, and Spring to validate the relevance and use of these recipes in open-source software.

- A **qualitative analysis** based on the same projects, to determine reasons for cases in which recipes get violated.

In particular, we want to highlight three general findings, besides the specific ones described later in the paper: (1) for each source code contribution it is important to state its intent, tests, and dependencies, (2) open-source projects largely follow best-practices from industry, and (3) while compliance to these principles is generally substantial, there are cases where deviations make sense from a software developer's or project manager's point of view.

The remainder of this paper is structured as follows. In Section 2.2, we introduce our research design, followed by detailed description of the research process and its results in Sections 2.3 and 2.4. In Section 2.5, we discuss the results followed by the most relevant related work in Section 2.6. The threats to validity of our research are summarized in Section 2.7. Finally, we conclude with our main findings in Section 2.8.

2.2 Research Design

Our work addresses the following two research questions:

RQ1: *What recipes can support software managers during the integration of source code contributions?*

RQ2: *To which extent do open-source projects follow source code integration recipes, such as those proposed in this work?*

Our approach to address these questions follows a two-step research design, as illustrated in Figure 2.1. The first step is an interview of software managers to collect best-practices for source code integration. In a second step, we evaluate to what extent the proposed recipes can be identified in open-source projects, and in which situations open-source developers deviate from the proposed best-practices. For this step, we mined artifacts from 21 open-source

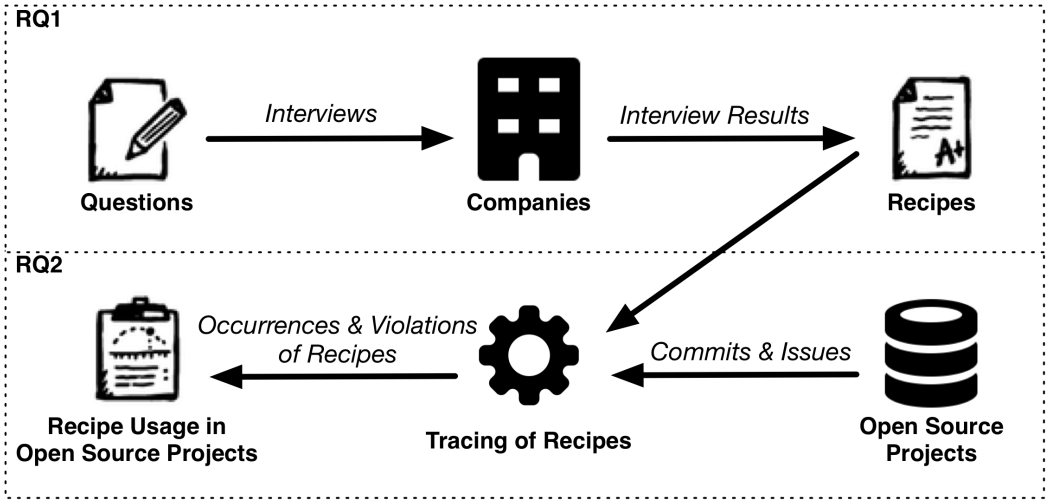

Figure 2.1: Approach Overview

projects to analyse their recipe compliance, and we qualitatively investigated cases of non-compliance.

2.3 Pragmatic Code Integration

In the first step of our research, we interviewed software managers from industry to discover best-practices for the integration of source code contributions into a code base.

2.3.1 Methodology

We contacted software development departments of seven international companies with headquarter or branches in Switzerland. Those companies were active within the banking, consulting, and software engineering sector. In a short summary, we presented the findings of our preliminary work [Brandtner et al., 2014] and asked them for participation. Three companies volunteered to participate in our interviews, and we scheduled interview sessions with three to four employees per company. The duration of an interview session per par-

ticipant was 30 minutes, and the sessions took place at the interview partner's offices. At the beginning of each session, we collected standard background information on the participant (see Section 2.3.2). We asked each participant for best-practices that are established in her company, and how she judges its importance for source code integration. Each interview session is documented with an audio record and semi-structured notes in a questionnaire style. For the design of the interview sessions we followed the guidelines of Kitchenham and Pfleeger [B. Kitchenham and S. Pfleeger, 2008]. The questionnaire was filled out by the interviewer in accordance with the interview partner.

2.3.2 Interview Partners

We interviewed eleven people from software development departments of three different international companies located in Switzerland. The average software development experience of our interview partners was 12.3 years. Each interview partner has a solid background in developing Java applications, and knows about the challenges of integrating source code contributions.

	Job Title	Experience	Company
P1	Software Manager	23 years	A
P2	Software Manager	10 years	A
P3	Software Manager	20 years	A
P4	Software Manager	8 years	A
P5	Lead-Developer	10 years	B
P6	Senior Developer	14 years	B
P7	Lead-Developer	12 years	B
P8	Senior Developer	11 years	C
P9	Senior Developer	10 years	C
P10	Lead-Developer	10 years	C
P11	Software Manager	7 years	C

Table 2.1: Summary of interview partner's job titles and professional backgrounds

Half of the interview partners work in multiple projects concurrently. None of the interview partners worked in the same project as one of the other interview partners. Table 2.3.2 lists the self-described job title, the software development experience, and the company affiliation of our interview partners.

2.3.3 Interview Results

In the following, we list the resulting recipes R1-R3. Each recipe consists of a rule (the actual guideline), ingredients (artifacts or data), variations (for project-specific requirements), a purpose (why a recipe is useful), and a description based on the results of the interviews.

R1 – Reveal the Intent.

Reveal the Intent	
Rule:	Contributions should contain at least one link to an issue
Ingredients:	Commit message, kind of commit (e.g., code, merge, etc.), number of changed resources per commit
Variation:	R1.1: *Small* contributions should contain *either* a link to an issue *or* a useful description
Purpose:	*Release management* – documenting which issue a source code contribution fixes

Our interview partners rated this recipe at least as *important* to keep track of what concrete bug or change request a code contribution actually solves. This eases the task of deciding whether a given contribution should be part of a new release. Some participants shared a less restrictive perspective on this topic. They claimed that the effort of creating a new issue entry even for small source code change is too large:

> *"For smaller commits, the creation of an issue may last too long. In such a case, a description of the problem and the solution directly in the commit message is preferred."* –P4

Hence, we have established a variation **R1.1** of the basic recipe to cover the exceptional case of small commits. All interview partners used the number of changed resources to define commits that qualify as "small", but we were not able to find a common definition as the mentioned values vary between one and up to ten changed resources, depending on the changed resource type. The ingredients of this recipe are the commit data from the VCS. Depending on the project, the interpretation of this recipe varies in the definition of a small commit, and the reference key or description style used in the commit message.

R2 – Align the Tests.

Align the Tests	
Rule:	Changes in source and test code should be combined in one commit
Ingredients:	Ratio of source and test code changes in a commit
Variation:	R2.1: Commits that contain only source code changes should be proceeded or followed by a commit containing test code changes
Purpose:	*Quality management* – ensuring testing of source code contributions

This recipe states that source code contributions and tests should be commited together. This is rated as important or very important, depending on the degree of existing test coverage within a software system:

> "*Such a recipe is only important for new projects and existing projects with a good test coverage.*" –P2

One reason mentioned multiple times is the effort of the preparation work, which would be needed to create meaningful test cases in a project that is currently not widely using a test environment. For example, the creation of a test case, which relies on data from a database, requires mocks [Taneja et al., 2010] or a testing database. In such cases, the preparation effort is higher than the actual code change in a software system. A further impact on this recipe raised by our interview partners are company-specific modifications of test-driven development. For example, in some projects the test code changes get committed before the actual source code change in case of a bugfix but not in case of a new feature.

> "*It is important to ensure the contribution of test code, but not every commit addressing an issue contains source code and test code changes.*" –P9

We established a variation **R2.1** of the basic recipe to cover situations in which source code and the test code changes get committed in different commits. The ingredient of this recipe is the commit data, especially the list of changed resources from the VCS. Depending on the test setup in a software project, the interpretation of this recipe varies on the ratio between source code changes and test code changes within one commit. For example, in a project with a high test coverage, a change in the source code might led to a relatively small change of the test code, whereas in a project with a low test coverage, a small source code change may lead to a large change in the test code.

R3 – Reveal the Release Dependencies.

Reveal the Release Dependencies	
Rule:	Commits that address multiple issues should be marked as such
Ingredients:	Commit message Keys of linked/duplicated issues
Variation:	None
Purpose:	*Release management* – isolation of source code changes

The isolation of source code changes, addressed in this recipe, is important as it allows software managers to cherry-pick different feature commits and bugfixes for a release. Some interview partners address the isolation of source code changes by allowing only one issue key per commit message.

> "*We enforce commit messages with only one issue key and use dummy issues to encourage developers to not include issue independent changes (e.g., typos, etc.) in a commit.*" –P5

Others allow multiple issue keys per commit message in case that a commit addresses multiple issues. Alternatively, a developer may reference one issue in the commit message, and mark the others as duplicates of the referenced one.

The ingredients of this recipe are the commit data from the VCS and the data of resolved issues from the issue tracker. Depending on the project, duplicated or related issues get explicitly linked in the issue tracker, and have multiple issue keys in the commit message. For example, in case of a resolved issue without a commit in the VCS (issue key is never mentioned in a commit message) a link to another issue should exist in the issue tracker.

2.4 Analysis and Results

In the second step of our study, we validated those recipes based on open-source project data. We mined projects from the ASF, JBoss, and Spring in order to trace occurrences of the proposed recipes in the wild. We extracted data of a six months period (July 2014 to December 2014) from the version control system (VCS) and the issue tracker hosted by the projects.

2.4.1 Project Selection and Data Extraction

We investigated projects from well-known open-source communities, such as ASF, Eclipse, JBoss, or Mozilla. Initial analysis showed that most of the projects of the mentioned communities use Git as VCS, or at least offer a Git mirror of the SVN repository. JIRA is used as issue tracker by most of the projects within the ASF and JBoss community, and Bugzilla is used by most of the projects within the Eclipse and Mozilla community. Furthermore, we looked at the used programming language and the build automation systems used within the different open-source projects. We decided to analyze Java-based open-source projects from the ASF, JBoss, and Spring community, primarily due to their active development history. We used the platform OpenHUB.net to search for Java projects from the ASF, JBoss, and Spring community under active development. Based on this search, we randomly selected 60 Java projects that use Maven from the ASF (40), the JBoss (10), and the Spring (10) community. We discarded project with less than 120 created or resolved issues, and less than 120 commits in the analyzed time-period of six months (on average 20 commits/issues per month).

The remaining 21 projects are listed in Table 2.4.1. One of the largest projects in terms of development activities is the Apache Ambari project, with 2577 commits from 39 contributors and 2450 issue changes (created/resolved). Based on these numbers, a committer of the Apache Ambari project contributed on average 66.08 change sets in the second half of 2014. We developed a Java application to extract the source code history and issue data used for our

Project	Commits	Issues	Committers
Apache			
Accumulo	630	403	17
ActiveMQ	271	179	12
Ambari	2577	2450	39
Camel	1669	591	44
CXF	805	294	16
Drill	435	611	13
Felix	353	200	17
HBase	998	1374	29
Hive	1310	1663	25
J.rabbit-Oak	801	360	15
Karaf	272	354	9
PDFBox	936	616	6
Sling	1512	473	20
Spark	1617	2158	19
TomEE	510	184	6
JBoss			
Richfaces	388	329	12
WildFly	591	560	75
Windup	368	231	8
Spring			
Framework	942	703	20
Integration	165	167	6
XD	426	437	24

Table 2.2: Development activity overview of the 21 selected projects (July 2014 to December 2014)

further analysis. After extraction, we stored the data in a relational database. For the extraction of the Git data, we used the JGit[1] library, and the issue

[1] `JGit-http://www.eclipse.org/jgit/`

tracker data was extracted through the REST API of JIRA. An archive with the resulting data is available for download on our website.[2]

2.4.2 Recipes in Open-Source Projects

We use the extracted data for the further compliance analysis in this paper, which consists of a quantitative and qualitative discussion for each recipe.

Reveal the Intent

For the analysis of this recipe, we separate commits into two groups, *small commits* and *large commits*. In our context, we define commits as "small" if they address only one source code file and as "large" if they address multiple source code files. According to the interview results, small commits do not necessarily require an issue key in the commit message, but should have a description.

Quantitative analysis. We analyzed the commits of each group and looked for commit messages containing issue keys based on the JIRA issue key identifiers of each project. For example, the issue identifier for the Apache Camel project is *CAMEL*, and the format of an issue key (i.e., issue #1332) is as follows: *CAMEL-1332*. Additionally, we calculated the average commit message length within each commit group to find out if the absence of issue keys influences the length of the provided description in the commit message.

Result R1: 14 of 21 projects use issue keys in more than 75% of large commits.

	0-50%	51-75%	76-90%	91-100%
# of Projects	1	6	6	8

Two-thirds of the analyzed projects regularly use issue keys in commit messages of changes addressing more than one source code file. In the remaining

[2] `http://www.ifi.uzh.ch/seal/people/brandtner/projects/recipes.html`

third of the projects, we also found large commits with issue key. However, in these cases the issue keys were not used regularly (<76% with issue key). Especially in case of the Apache TomEE project, the usage of issues keys for large commits is not overly established (only 26% of contributions contain an issue key).

Result R1.1: 9 of 21 projects use issue keys in more than 75% of small commits.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	7	5	4	5

The results are different in case of small commits, 57% of the analyzed projects do not use issue keys for those commits on a regular basis (<76%). This is also reflected in Figure 2.2, which depicts the usage of issue keys in commit messages for the different change set sizes calculated across all analyzed projects and for a selection of projects.

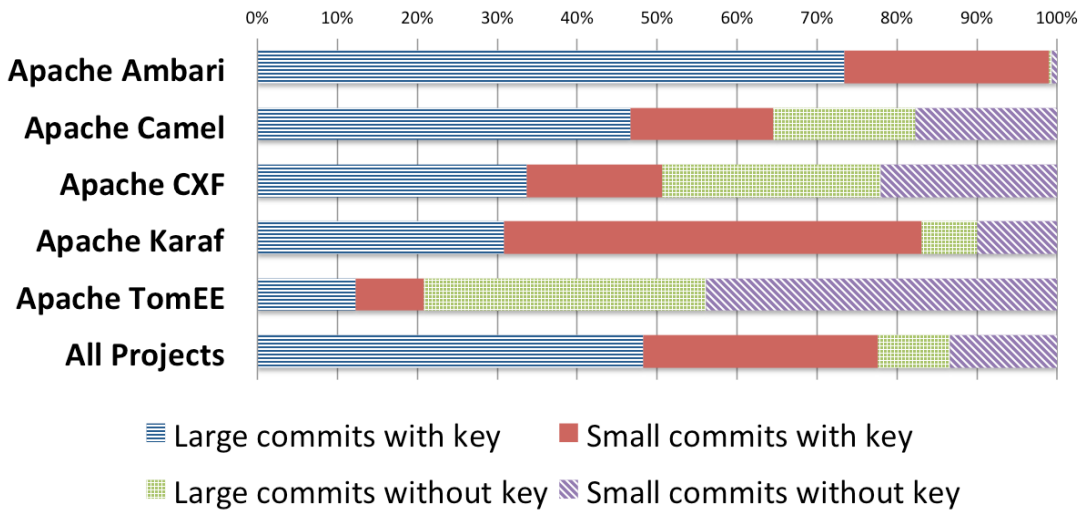

Figure 2.2: Commits with and without an issue key in the commit message

In respect to variation R1.1 of recipe R1, we analyzed also the average

commit message length. The analysis showed that the average message length of commits without an issue key is up to 69% shorter (e.g., Apache Felix) than those of commits with issue key. Only in case of the Apache Ambari project, the commit message of small commits without an issue key was on average 12% longer compared to the message of small commits with issue key.

Qualitative analysis. In order to conduct a further qualitative analysis of cases when issue keys have been omitted, we decided to investigate a randomly selected subset of five commits per project without key. We filtered source code commits without issue key and categorized the resulting 42 commits according the described change in the commit message. The categories are: build failure (11.9%), dependency management (19%), rollback (7.1%), versioning (35.7%), and other changes (26.3%).
Build failure. Problems related to the build process of a software system are often mentioned in commit messages without an issue key. For example, in the Apache Accumulo project: *"unbreaking build; trivial change"* or the Apache PDFBox project: *"added rat exclude rules to avoid build failures"*. As indicated by the commit messages, those commits usually not not contain large changes.
Dependency management. Version upgrades of used libraries or similar changes are potential candidates for commits without an issue key. For example, in the Apache Jackrabbit project: *"use latest H2 version [...]"*. Those kind of commits often only consist of changes to the *pom.xml* files of Maven-based software projects.
Rollback. Commits that are completely or partly reverted to a previous state are candidates for commits without an issue key. For example, in the Apache Hive project: *"Rolled back to 1643551"*. As shown in the example, the message of a rollback commit often contains only a reference in the VCS but no reference to the affected issue in the issue tracker.
Versioning. The release of a software system includes the change of the according versioning information in the software artifacts. Some of the analyzed projects (e.g., Apache Felix, or Apache Sling) use the Maven release-plugin for this step. The default commit message generated by the Maven release-plugin starts with *"[maven-release-plugin]"* and does not contain an issue key.

Other changes. Other changes that often do not contain links to issues are the polishing of code snippets, the addition of missing files, and typo fixes.

Summary. We were able to show that most of the analyzed projects (66.7%) comply with recipe R1 to a compliance level of at least 75%. However, each of these projects has a residual of commits that deviate from this recipe.
Exceptions. Fixing build failures, dependency management, rollbacks, or versioning are exceptions that cause violations of this recipe. Those changes are not directly linked to any concrete issue, and hence can do without explicit link. Most importantly, missing links of these changes are typically not detrimental to the stated purpose of this recipe (release management).

Align the Tests

For this analysis, we established four groups for the classification of commits based on the affected resources: (1) source code files, (2) test code files, (3) source code and test code files (combined commits), and (4) other files. This classification is inspired by the structure of the Maven-based projects used in our analysis. Any change in a resource located under *src/main* is classified as *source code* change. A *test code* change is any change of a resource located under *src/test*. Every resource changes outside one of the two mentioned locations is classified as *other* change. For the association of source and test code changes, we depended on the naming schema of Maven. The test code for a class in the *src/main* directory has to resist under the same relative path in the *src/test* directory with a *Test* suffix in the resource name. For example, the test code for a class *src/main/ClassA.java* is located under *test/main/ClassATest.java*.

Quantitative analysis. We analyzed the structure of source code, test code, and combined commits of each project. As many of the analyzed projects are organized through modules, we had to take care of cascading directory structures, which means multiple *src/main* and *src/test* directories within a repository.

Result R2: In 1 of 21 projects more than 75% of the commits contain source code and test code changes.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	17	3	1	0

The results of the analysis showed that only in case of the Spring Integration project, the majority of the source code changes (79%) get committed together with test cases. An average value of 30.7% (across all projects) indicates that combined source code and test code commits do exist, but they are not the largest group of commits existing in open-source projects. The largest group of commits are pure source code commits with an average value of 53.4% (across all projects). The group of pure test code commits reach an average value of 15.9% (across all projects).

Result R2.1: In 14 of 21 projects more than 75% of the source code commits have test code commits.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	3	4	10	4

In many cases, source code and test code changes end up in the VCS as independent commits. Addressing the amount of independent source code and test code commits, the compliance is better compared to the compliance of the combined commits. 14 of 21 of the analyzed projects have a similar amount of pure source code and test code commits resulting in a high compliance level (>75%). For example, the Spring Integration project in Figure 2.3.

However, the shares computed across all projects indicate the there are projects which do not follow this recipe. One example for such a project is Apache PDFBox. The majority of commits (87%) in the VCS of this projects are pure source code changes and only 5% of the commits are combined ones.

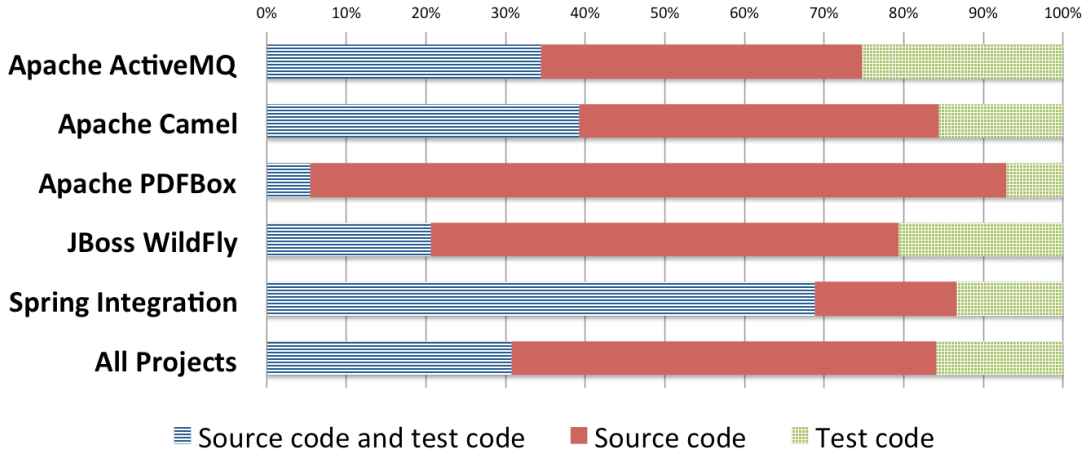

Figure 2.3: Share of resources affected by commits

Such a large amount of pure source code commits can be an indication for shortcomings in the testing process.

Qualitative analysis. In order to dig deeper into why source and test code is not commited together, we selected all issues that have at least one comment containing the term *"unit test"*. Most of the comments in the resulting 1632 issues do not address the missing testing of a contribution. Therefore, we manually inspected 50 issues and found three issues that are affected by this recipe.

The found issues indicate that the management of open-source software projects take care of test code contributions through patches. For example, a patch for issue SLING-4212 of the Apache Sling project was not integrated and commented with: *"[...] add unit tests for the case name = null"*. A similar comment can be found for issue SLING-4112: *"open points: review if we should add some more specific unit tests for the new classes [...]"*. Again, the according source code was not directly integrated into the codebase. A slightly different example was found in the Apache Jackrabbit-Oak project (OAK-2301): *"[...] That needs to be done, plus unit tests."* Despite this comment, the patch was integrated into the codebase without any tests. Such a behavior can be an indicator for the performance differences between recipe R2 and its variation

R2.1 in the qualitative analysis, as test code changes get integrated at another point in time as the according source code changes (or not at all).

Summary. We were able to show that many of the projects (66.7%) comply at least with R2.1, a variation of recipe R2. However, depending on the project, this recipe sometimes gets violated or is not followed at all.
Exceptions. A typical exception of this recipe is the integration of a patch without taking care of missing tests.

Reveal the Release Dependencies

In this analysis, we focused on issues of the types *bug* or *feature* exclusively, as we expect a commit in the VCS for these types. The resulting set of issues was split into three groups: (1) issues addressed in an exclusive commit, (2) issues addressed in bulk commits, and (3) issues without commit. An exclusive commit in this context is manifested through a commit messages that contains exactly one issue key and an optional description. A bulk commit is defined in our context as commit that contains multiple issue keys in the commit message.

Quantitative analysis. We analyzed the structure of issues that have been marked as resolved based on changes in a bulk commit and resolved issues without commit.

Result: In all projects more than 75% of the commits that address multiple issues were marked as such.

	0-50%	51-75%	76-90%	91-100%
# of Projects:	0	0	7	14

The results showed that commits addressing multiple issues at once exist in all of the analyzed projects. Each project takes care of marking such commits with an issue key for every affected issue.

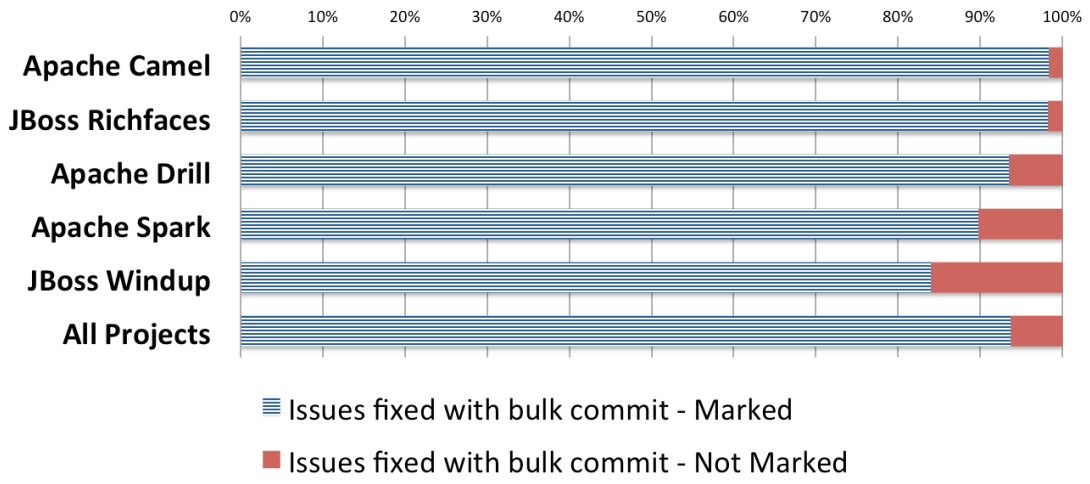

Figure 2.4: Resolved issues with and without dedicated commits

Figure 2.4 depicts the issue shares calculated across all projects and a selection of five further projects. The overall share of issues fixed with a bulk commit are marked as such on average in 93.7% of the cases. This can be seen as a strong indication for the existence and the enforcement of this recipe in open-source projects.

Qualitative analysis. For the qualitative analysis, we decided to investigate the rare cases in which an issue was marked as resolved without an associated source code change. We manually investigated a subset of 50 resolved issues without an associated commit. In this subset, we found five issues that violate recipe R3. One of these issues is FELIX-4654, which seems to be resolved by the code change of another issue. However, there is only a code snippet of the latest development snapshot mentioned in the comment. Based on the information in the issue tracker, it is not possible to determine the related issue or commit. A similar case is issue AMQ-5313 of the Apache ActiveMQ project. This issue was also fixed by a change that addresses another issue. Instead of linking the according issue tracker entry, the developer describes the changed configuration and its effects in the issue comments. The result is an issue entry with plenty of text that is hard to understand for all involved stakeholders.

Another interesting case is issue KARAF-3218. This issue is marked as duplicate of issue KARAF-3075, which addresses the underlying problem that causes both issues. However, the difference of issue KARAF-3075 and KARAF-3218 is the affected version of Karaf. In KARAF-3218, the issue is raised for version 2.3.2 and in KARAF-3075 it is raised for version 2.3.5 and later. To solve issue KARAF-3075, the code change was initially applied to the branches of version 2.4.x and later. The situation was clarified with issue KARAF-3417, where the reporter of issue KARAF-3218 asked for a backport.

Summary. We were able to show that all of the projects comply with recipe R3 to a compliance level of 75% or more. However, each of these projects has a small residual (on average $<$7%) of commits that violate this recipe.
Exceptions. Issue entries describing side-effects caused by other issues are exceptions that violate this recipe. Such entries have in common that no commit is associated to them, but the comments contain clear indications that the issue disappeared with a code change that affects another issue.

2.5 Discussion

The results of our interview study in industry as well as the quantitative and qualitative analysis of open-source projects indicate that software projects largely follow the proposed recipes for release and quality management. Table 2.5 summarizes the pragmatic recipes, the median compliance level, and the number of projects that fulfill a certain recipe on a *compliance level of 75% or more*.

In the following, we discuss the analysis results of the proposed recipes according to their purpose.
Feature and release management. Two of the proposed recipes (“R1 – Reveal the Intent” and “R3 – Reveal the Release Dependencies”) address the feature and release management of the source code integration process. For both

	Recipe	Median	>75%
R1	Contributions should contain at least one link to an issue	81.6%	14
R1.1	Small contributions should contain either a link to an issue or a useful description	57.2%	9
R2	Changes in source and test code should be combined in one commit	40.2%	1
R2.1	Commits that contain only source code changes should proceed or followed by a commit containing test code changes	77.8%	14
R3	Commits that solve multiple issues should be marked as such	93.7%	21

Table 2.3: Median compliance level and number of projects with a compliance level greater than 75%

recipes, the declaration of the commit content and the correct association to an issue plays an important role.

Especially before code freeze deadlines, software managers have to handle large amounts of source code contributions. It is impossible for them to inspect the content and the purpose of each single source code contribution. Time constraints caused by, for example, release plans force software managers to decide whether or not to integrate a contribution within a few minutes. A clear declaration of a contribution's content with an issue tracker entry can support the decision making and reduce the risk of integrating less important or even unneeded changes into a code base.

The interviews as well as the analysis showed that projects follow these recipes to a large extent. Despite the rather high compliance rate of the analyzed projects, it is still possible to find violations for each of the recipes. A typical reason that we found for such a violation is the contribution of source code that addresses a small change (e.g., typo fix). The violations which we found in the open-source projects are in line with the interview results of allowing small code changes without issue entry. During the interviews, the

relatively high effort of opening issues was mentioned as a hurdle to having issue keys even for small changes. However, it remains unclear whether the effort saved when implementing a source code change without issue key is more substantial than the additional effort caused by the absence of a clear purpose during the feature integration, especially considering that the time of a software manager is often more costly than that of a developer. Of course, there are cases in which the creation of an issue entry can indeed be seen as overkill. For example, a version change of a dependency in a Maven-based software project is achieved by simply replacing the old version number through the new one in the *pom.xml* file. Creating an issue entry for every such case is usually not important, especially given that the implications of such changes for feature integration are limited.

Exceptional cases are source code contributions that address multiple issues at once. The analysis showed that those contributions contain a reference to at least one issue entry, but there is no guarantee that each affected issue is indeed revealed. The absence of a reference to an issue entry that is addressed by a commit can be caused by various reasons. In our analysis, we were able to find at least two different cases in which an issue was not referenced by the issue resolving commit. In one case, the issue seemed to be caused by a side-effect of another issue. As soon as the underlying issue was resolved, the side-effect disappeared. However, we were not able to verify the potential issue relationship based on the data available in the VCS and the issue tracker. Especially the textual description of potential relationships mentioned in the comments of the according issues without mentioning an issue number makes it impossible to verify relationships. In another case, the wrong classification of an issue as duplicate of an existing one led to a situation in which a bug fix was applied to a subset of affected versions only. Therefore, the absence of the issue reference in the commit message was correct even if the commit contains the actual fix. In this case the failure occurred on the issue management level and not on the source code integration level.

Contribution quality management. One of the proposed recipes (“R2 – Align

the Tests") addresses the internal quality of source code contributions and its impact on the integration process.

An important goal of software managers is to get new features implemented and bugs fixed within an appropriate time span. This time span varies depending on the project and the priority of the underlying issue. An important attribute in such a scenario is the quality testing of a source code contribution to ensure the expected behavior of a software system after integration. The interviews as well as the analysis showed that many projects follow this recipe, but the level of compliance is strongly influenced by the history of a software project. For example, it is more likely that a software project, which followed this recipe from the beginning, continue following this recipe during the remaining development life cycle. We were not able to find a project that started off with no or very low test coverage, but started to strictly follow R2 at a later phase of the project. Those findings are in line with the results of the interview study. All of our interview partners confirmed that starting following such a recipe in a project only makes sense at the beginning. The major challenge in all of the projects was to reach a solid testing density across the whole system. A high density of tests for single modules or classes is nice to have, but does not necessarily reduce the risk of problems at the system at-large. For example, a high testing density of a module that represents one percent of the whole code base cannot reduce the risk of a problem in the remaining ninety-nine percent of the code. Of course, this also depends on the criticality and error-proneness of the module in a software system. It is likely that a high test density for a central and error-prone module can reduce the risk of a failure more efficiently than a high test density of a rarely used utility module.

Our analysis showed that only in a few cases, all projects violate variation R2.1, but such violations occur less often than for R2. We assume that a violation of this recipe is strongly correlated to the importance and the size of a code change. For example, we found cases where software managers commented that the code change was integrated without code changes, but the issue remains open as long as there is no test case contributed. Most according

issues were classified as bugs and had at least a *major* priority, supporting our theory that this is particularly prone to happen for high-impact bugs for which the software manager quickly needs a fix and is willing to live with quality risks to have the problem solved. In a small number of other cases, software managers integrated a source code change and wrote a comment that the tests are missing, but nevertheless marked the issues as resolved. The according issue entries for these cases were often classified as minor improvements with a low priority.

In a further investigation, we found indicators that the number of test code changes indeed impacts the test coverage of a software project. For example, the quantitative analysis results of the Apache PDFBox project indicate a rather low number of changes in the project's test code. A public available quality analysis of the latest PDFBox version[3] showed that the low number of test code changes is reflected in a rather low test coverage (18.7%) as well.

Finally, as a corollary of our study, we found that the *Hadoop QA* bot is used in multiple of the analyzed projects (e.g., Ambari, HBase, Hive). This bot automatically conducts a number of heuristic quality checks for every submitted source code contribution, two of which concern testing. The first such check is the number of new or modified tests (e.g.,*"+1 tests included. The patch appears to include 3 new or modified test files"*), the second check addresses the test results (e.g., *"+1 core tests. The patch passed unit tests in ambari-server."*). We speculate that such a bot can be beneficial, especially for the decision making of a software manager during the integration process. For example, for a quick check if test cases have been changed, it is no longer necessary to dig into the source code of every commit with such tooling.

[3] `http://nemo.sonarqube.org/dashboard/index/332186` [accessed: Jan 26, 2015]

2.6 Related Work

The integration of source code contributions and issue management are often discussed topics in literature. We divide the field of the related work into two areas: source code contribution management and issue management.

Source code contribution management: Tsay et al. [Tsay et al., 2014] investigated source code contributions in pull-request-based environments, such as GitHub. They analyzed different aspects for the rejection or the integration of source code contributions into the main code base. The results of their research showed that the integration of smaller contributions is more likely compared to large source code changes affecting multiple files. Pham et al. [Pham et al., 2013] investigated the test culture in social coding platforms and found out that the inclusion of test cases can raise the likelihood for integration as well. Gousios et al. [Gousios et al., 2014] additionally investigated issue descriptions in social coding platforms, and showed that an insufficient task articulation is the major rejection reason for source code contributions. The research on branching in software projects addresses source code integration as well. However, branching is a different paradigm, which supplements the pull-request-based paradigm. The investigation of related work on branching addresses aspects, such as reasons for the creation of a branch, the optimal points in time for the merge of a branch [Bird and Zimmermann, 2012], and the impact of structural characteristics within a branch onto the source code quality after a merge [Shihab et al., 2012]. A third way to contribute source code to a software projects is the submission of patches via a mailing list or an issue tracker. Research on the handling of patches addresses the reviewing and patch integration process in open-source projects, such as the Apache Server project (e.g., [Mockus and Votta, 2000]) or the Linux Kernel (e.g., [Jiang et al., 2013]). The investigations of Brun et al. [Brun et al., 2011] showed that, for example, a high number of changes addressed within a patch negatively influences the integration process.

Issue management: Anvik et al. [Anvik et al., 2006] investigated an approach for a semi-automatic bug assignment based on a machine learning

algorithm. They used the issue assignee-field and status-change data from the issue tracker of the Eclipse and Firefox project for their predictions. The problem they faced is that every project tends to use these fields differently (e.g., dummy assignee instead of a developer for each new bug, etc). Guo et al. [Guo et al., 2011] carried out a study in combination with a quantitative analysis to determine reasons for bug reassignments. A similar work with a focus on re-opened bugs was performed by Zimmermann et al. [Zimmermann et al., 2012]. Their results showed that the initial priority assignment and poor bug descriptions strongly influence the number of reassignments and reopening of issues. The issue assignment process can be supported by results from investigations on the classification of issues. Rastkar et al. [Rastkar et al., 2010] proposed an bug summary generator to support developers to quickly access experiences of other developers within the same project, which can give an insight for the issue assignment as well. Giger et al. [Giger et al., 2010] proposed a prediction model to estimate the expected time needed to resolve a certain bug. This prediction may help with issue prioritization. However, the issue assignment process and the proposed predictions preferable work with complete bug descriptions. Aranda and Venolia [Aranda and Venolia, 2009] investigated coordination patterns in issue trackers during bug fixing. One finding of their work is that bulk data changes in an issue tracker (e.g., status change of multiple issues) may negatively influence coordination patterns. Bettenburg et al. [Bettenburg et al., 2008] conducted a study with developers and bug reporters to find out what information the developers expect in a bug report to measure the quality of a bug. They found out that there is often a mismatch between the provided information of users and the information needed by developers. These results align with a study of Ko and Chilana [Ko and Chilana, 2010], which concludes that the major impact of Mozilla's public issue tracker onto the development process are not the bug reports, but the hiring of talented developers. According to the work of Breu et al. [Breu et al., 2010], it is important to keep issues up-to-date with, for example, comments. Such status updates can effectively engage all involved stakeholders and support the project management.

To the best of our knowledge, there is no work that investigates a pragmatic source code integration approach for open-source projects based on best-practices from industry.

2.7 Threats to Validity

Interviews and empirical studies have limitations that have to be considered when interpreting the results of our work.

Threats to external validity. These threats address the ability to generalize our results for software projects. The relatively low number of interview partners limits the ability to generalize the proposed set of recipes for software projects across different domains or companies. Further interviews with interview partners from other companies or domains may help to overcome this limitation. In our analysis, we used merged data from Java-based open-source projects of three open-source communities. We ignored differences in the project maturity and size during the analysis. Therefore, projects with more development activity may influence the average results more than projects with low activity. Furthermore, the results of our analysis are restricted to software projects that use Java as programming language, JIRA as issue tracker, and Maven as build system.

Threats to internal validity. These threats address the ability to draw conclusions based on our interview results. The use of an example set of draft recipes may introduce a bias in the interview results. We assume that the bias does not affect the results of the importance rating, as it reflects the personal opinion of each interview partner about the proposed statements. Another threat is the potential survival bias caused by the use of data mined from the VCS and the issue tracker. Especially the data of the VCS represents only cases of source code contribution that have been accepted but misses those, which have been rejected. We tried to mitigate this threat by focusing on patch contributions uploaded to the issue tracker. In difference to the VCS, the issue tracker stores the whole history of an issue and the contributed patches, independent of the decision to accept or reject it.

2.8 Conclusions

The analysis of 21 open-source projects showed that the integration of source code contributions in software projects largely follows recipes for feature, release, and quality management.

In this work, we propose an initial set of three pragmatic recipes plus variations for source code integration based on best-practices of software managers from industry. Our proposed recipes cover different aspects of Continuous Integration, from feature and release management to quality management. The proposed pragmatic recipes do not influence the technical process to integrate source code (e.g., merging, building, etc). Instead, these recipes enable a software manager to, for example, find source code contributions, which are not ready to be included into a release. Further, we analyzed to what extent open-source projects also follow similar recipes. After studying 21 projects, we concluded that most of the identified recipes are wide-spread in the open-source community as well. The implications of our work for researchers and software managers are:

- Raise the awareness about the importance of the source code integration step.
- Reveal the intent, tests, or dependencies of a source code contribution to save value working time.
- Record the measurements and compliance (e.g., test coverage, code style) of each source code contribution.

Those implications can be directly applied in modern build and release management systems. For example, in some projects we found a lightweight approach called *HadoopQA* bot, which addresses some aspects of the listed implications.

Further, we need to explore more recipes for source code integration, both in industry and the open-source community. These additional recipes should not be limited to source code contributions and issues, but should also include

other Continuous Integration data sources and artifacts. Our ultimate goal is to provide a collection of established best-practices, easing the adopting of Continuous Integration in practice.

3

Discovering Loners and Phantoms in Commit and Issue Data

*Gerald Schermann, **Martin Brandtner**,*
Sebastiano Panichella, Philipp Leitner, and Harald Gall
Published in the 23nd International Conference on Program Comprehension, 2015
DOI: 10.5167/uzh-110093

Abstract

The interlinking of commit and issue data has become a de-facto standard in software development. Modern issue tracking systems, such as JIRA, automatically interlink commits and issues by the extraction of identifiers (e.g., issue key) from commit messages. However, the conventions for the use

of interlinking methodologies vary between software projects. For example, some projects enforce the use of identifiers for every commit while others have less restrictive conventions. In this work, we introduce a model called *PaLiMod* to enable the analysis of interlinking characteristics in commit and issue data. We surveyed 15 Apache projects to investigate differences and commonalities between linked and non-linked commits and issues. Based on the gathered information, we created a set of heuristics to interlink the residual of non-linked commits and issues. We present the characteristics of *Loners* and *Phantoms* in commit and issue data. The results of our evaluation indicate that the proposed *PaLiMod* model and heuristics enable an automatic interlinking and can indeed reduce the residual of non-linked commits and issues in software projects.

3.1 Introduction

The interlinking of commit and issue data plays an important role in software development, during the release planning or the bug triaging. It has become a de-facto standard in software projects, which is reflected in guidelines of large open-source communities, such as Apache: "*You need to make sure that the commit message contains [...] and ideally a reference to the Bugzilla or JIRA issue where the patch was submitted.*" [Apache Software Foundation,].

However, sometimes developers violate guidelines and commit changes without issue reference [Bachmann et al., 2010, Romo et al., 2014]. For this reason, various research studies investigated possible ways to recover missing links (e.g., [Bachmann et al., 2010, Fischer et al., 2003a, Aranda and Venolia, 2009]). Researchers proposed heuristics (e.g., [Bachmann and Bernstein, 2009, Mockus and Votta, 2000]) to automatically recover missing links by mining issue tracking platforms and commit messages in logs of version control systems (VCS). These heuristics rely on keywords, such as "*bug*" or "*fix*", and issue ids, such as "*#41*", in commit messages. However, these approaches are often not sufficient to detect all of the missing links between issues and commits. Recent research (e.g., [Nguyen et al., 2012, Wu et al., 2011]) defined

more complex approaches based on text-similarity, which are able to recover a higher percentage of missing links. The former research has in common that it is build on a scenario where no explicit interlinking of commit and issue data is available. In difference to modern issue tracking systems, such as JIRA, which support an automated interlinking based on issue keys in commit messages. Hence, it seems valuable to take this information into account rather than starting from no links. The availability of an explicit interlinking enables new research directions. For example, the profiling of developers based on their activity in the VCS and the issue tracking platform [Brandtner et al., 2015b].

In this work, we investigate the characteristics of data interlinking in development environments that use a modern issue tracking platform with interlinking capabilities. For our investigation, we introduce a model called *Partial Linking Model (PaLiMod)* to support the integration of commit and issue data. On top of this model, we surveyed the interlinking of commit and issue data within 15 Apache projects that use JIRA as issue tracking platform. Based on the gathered information, we derived the characteristics of two interlinking scenarios called *Loners* and *Phantoms*. Loners in the context of our work describe single commits that have no link to the addressed issue. In case of a Loner, no other commit addresses the same issue, which is the major difference to Phantoms. Phantoms are commits without link occurring in a series of commits that address a certain issue. For example, a developer commits three changes addressing an issue, but only the last provides an issue key in its commit message. Based on the investigated characteristics, we propose heuristics to automatically detect *Loners* and *Phantoms* for reducing the residual of non-linked commit and issue data.

The main contributions of this paper are as follows:

- *A formal model to investigate the partial interlinking of commit and issue data.*
- *A survey to investigate the practice of commit and issue interlinking in 15 Apache projects.*
- *A set of heuristics to automatically handle missing links in partially linked commit and issue data.*

- *A prototypical implementation of the model and the interlinking heuristics.*

The results of our survey showed that on average 74.3% of the commits contain issue keys, but only 49.6% of the resolved issues are linked to a commit. Most of the linked commits (37.4%) are combined commits, which means they contain source code changes and test code changes. The largest group of the non-linked commits (47.0%) are those commits not addressing any source code change.

We evaluated the proposed heuristics in a series of simulated project scenarios that contain different residuals of non-linked commits and issues. For the simulation, we removed links between commits and issues that were explicitly linked by an issue key in the commit message. This evaluation approach is enabled by the PaLiMod model and allows for an accurate performance evaluation of the proposed heuristics.

The results of the heuristic evaluation showed that our approach can achieve an overall precision of 96% with a recall of 92% in case of the Loner heuristic, and an overall precision of 73% with a recall of 53% in case of the Phantom heuristic.

The remaining paper is structured as follows. We introduce the approach in Section 3.2, followed by an overview of the proposed PaLiMod model and the results of the survey in Section 3.3. In Section 3.4, we introduce the interlinking heuristics, and the evaluation in Section 3.5. We discuss results in Section 3.6. Section 3.7 covers the threats to validity. The most relevant related work is presented in Section 3.8. The paper is concluded with our main findings in Section 3.9.

3.2 Approach

The aim of this paper is the investigation of commit and issue data interlinking characteristics in software projects that have guidelines (e.g., [Apache Software Foundation,]) to link source code contributions to issues. Based on the findings, we derive a set of heuristics to enable an automatic interlinking of the

residual of non-linked commit and issue data within such software projects. For these investigations, we introduce a model to support the integration and analysis of commit and issue data. We investigate the extent to which developers follow such interlinking guidelines and the cases in which a guideline is not followed. The insights which we found are used to come up with a selection of heuristics to address cases in which an interlinking guideline was not followed.

We address the aim of our approach with the following two research questions:

RQ1: *What are the characteristics of interlinked commit and issue data in projects that have interlinking guidelines?*

RQ2: *How can an automatic interlinking of the residual of non-linked commit and issue data be enabled based on such characteristics?*

Figure 3.1 depicts the approach which we followed to answer the research questions of this work.

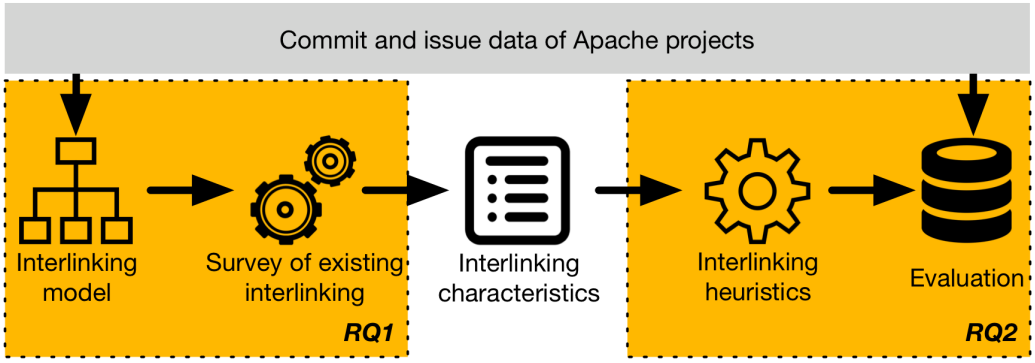

Figure 3.1: Overview research approach

In a first step, we defined the Partial Linking Model (PaLiMod), which is used to integrate and store the extracted commit and issue data for the project survey. Based on the integrated data of 15 Apache projects, we extracted a set of interlinking characteristics that occurred in all of the analyzed projects. In a

further step, the resulting characteristics of non-linked commits and issues are used to define heuristics to enable an automatic residual interlinking. Finally, we evaluate the performance of the proposed interlinking model and heuristics with a scenario-based approach.

3.3 Partial Linking Model & Project Survey

The proposed PaLiMod model is used for a project survey on the commit and issue interlinking in 15 Apache projects, which are listed in Table 3.3.

				Links	
ID	**Project**	**Commits**	**Issues**	**1:1**	**1:n**
P1	ActiveMQ	1058	901	165	166
P2	Ambari	5012	4249	2794	885
P3	Camel	3994	1272	161	610
P4	CXF	3792	968	25	137
P5	Felix	983	799	92	33
P6	Hadoop	1367	1998	62	125
P7	HBase	4110	4897	389	829
P8	Hive	2234	3926	1343	281
P9	Jackrabbit Oak	3781	1477	90	72
P10	Karaf	2059	1020	137	385
P11	PDFBox	1327	1100	99	120
P12	Sling	4174	1086	109	256
P13	Spark	5482	2748	452	706
P14	Stanbol	517	305	32	110
P15	Tika	332	347	53	24
	Mean	2681	1806	400	316
	Standard Deviation	1665	1412	716	287

Table 3.1: Development activity in Apache projects between September '13 and September '14 with number of commits, number of resolved issues, and the number of issues that are linked to one (1:1) or multiple commits (1:n).

3.3.1 PaLiMod - Partial Linking Model

The aim of the proposed PaLiMod model is to capture partly linked commit and issue data in software projects. The model relies on a graph-based schema to describe relationships of *Subjects*. A subject in the context of the PaLiMod model can be a change set, a change request, a person, or a resource. Some of the subjects have sub-subjects, for example, bug or feature to allow for a more precise classification of the according dataset. The categorizations are described in detail in Section 3.3.1 to 3.3.1. The subjects represent nodes in the proposed graph. The relationships of each subject are described by so-called *Annotations*. The proposed model offers annotations for relationships between commit, issue, person, and resource subjects. Figure 3.2 depicts an overview of the RDF based model with its subjects and annotations.

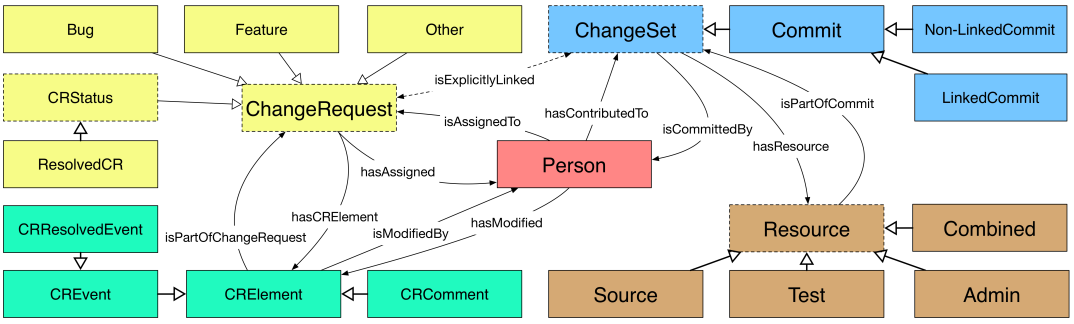

Figure 3.2: PaLiMod model

A major difference of the proposed PaLiMod model compared to existing approaches is the support of mixed environments that contain commit and issue artifacts that are partly explicit linked instead of environments without any linking. Additionally, the graph-based manner of the PaLiMod model enables the definition of multiple annotations between two subjects in concurrent.

The listed subjects and annotations represent an initial set of possible nodes and edges, which can be even expanded for other purposes as well. For example, by incorporating additional change request states in *CRStatus*, such as closed and reopened. Representing the state as an own subject instead of

a change request's property provides advanced querying options (multiple inheritance). A prototypical implementation of the PaLiMod model based on OWL/RDF is available for download on the project's website[1].

Classification of Commits

For the classification of commits, we propose the categories *linked* commits and *non-linked* commits. Linked commits are commits containing a reference (e.g., issue key) to an issue in the issue tracking system. Commits without such a mapping are called non-linked commits. Basically, a commit can tackle multiple issues at once, but state-of-the-art issue tracking platforms, such as JIRA, support a single issue key per commit message only. Thus, a single commit is either linked to a certain issue or not. On the other hand, a single issue can have multiple linked commits.

Classification of Resources

For the classification of resources, we propose the categories *source*, *test*, *admin* commits, and a *combination* of them (see Table 3.3.1). This classification is derived from the standard directory layout used in Maven projects.[2] Thus, this survey focuses on Java projects using Maven and following Maven's standard directory layout.

Classification	Description
Source	All associated resources are source code files.
Test	All associated resources are test code files.
Admin	All files are neither source, nor test code files.
Combination	The associated resources are a mixture of source code, test code, and other files.

Table 3.2: Resource Classification

[1] `http://www.ifi.uzh.ch/seal/people/schermann/projects/commit-issue-linking.html`

[2] `http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html`

A commit is classified as a source commit if all of its created, modified or deleted resources are source code files. A file is considered a source code file if its path in the project's directory structure contains the snippet *src/main/java*, which corresponds to Maven's standard directory layout. Similarly, the snippet *src/test/java* is used to identify test code files. Therefore, a commit is called a test commit if all of its addressed files are test code files. All other files, which are neither source, nor test code files are called admin files (e.g., changes in the pom.xml). Hence, a commit with only such admin files is referred to as an admin commit. Finally, commits which contain a mixture of source, test and admin files are summarized in the combination category.

Classification of Issues

We categorize issues based on the issue type information extracted from the issue tracker. As different naming schemes are used among the analyzed projects in respect to issue types, we assigned them to the simplified categories *Bug*, *Feature*, and *Other*. For example, issues with types *Bug* and *Epic* were assigned to the *Bug* category, in addition *Enhancement* and *Wish* are considered for the *Feature* category. Table 3.3.1 provides the complete mapping of issue types to the categories used throughout this work.

Classification	Description
Bug	All issues with associated issue type *Bug* or *Epic*.
Feature	All issues with associated issue type *Enhancement*, *Wish*, *Task*, *Feature Request*, *Subtask*, *Improvement*, or *Feature*.
Other	All remaining issues.

Table 3.3: Issue Classification

3.3.2 Apache Project Survey

In our project survey, we use data from VCSs (Git) and issue tracking platforms (JIRA) of 15 Java-based Apache projects (see Table 3.3). All projects are charac-

terized by a high activity in the VCS and the issue tracking platform within the last year. We first selected projects with high commit activity provided by OpenHUB.[3] In a second step, we looked at the projects' issue trackers and reduced our selection to 15 projects with high activity in both domains. A project's activity is considered high if there are at least 20 commits and 20 issues per month. We extracted commit and issue data of the period between September '13 to September '14 and stored it into instances of the PaLiMod model.

Survey Results

Table 3.3.2 shows the distribution of linked and non-linked commits of the analyzed projects. In addition, the shares of (sub-) categories source, test, admin, and combination are provided below the overall linked and non-linked shares. The ratio section in the table provides the ratio of non-linked to linked, including the sub-categories as well. For example, the portion of linked source commits is twice as high (ratio 0.5) as the portion of non-linked source commits in the Apache ActiveMQ project (P1). In the same fashion as Table 3.3.2, Table 3.3.2 shows a distribution between linked and non-linked as well, but from the issues perspective. Again the shares of (sub-) categories are listed below the overall linked and non-linked results alongside with the ratio information.

The survey of Apache projects showed that interlinking of commit and issue data is used by all of the analyzed projects. In the majority of the analyzed projects (12 of 15), the number of commits linked to an issue is higher than the number of commits without link. However, this result is not reflected in the number of linked issues, as the majority of issues is not linked to a commit in the analyzed projects (8 of 15). One reason for such a deviation is the existence of issue entries (e.g., tasks) that do not require a change in the VCS. However, an indicator against this assumption is the rather high amount of non-linked *Bug* issue entries. Another reason for the missing one-to-one mapping can be the circumstance that an issue fix requires more than one commit and thus,

[3] `https://www.openhub.net/`

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Linked	**64.0**	**98.3**	**60.6**	**44.4**	**48.6**	**84.1**	**90.0**	**95.2**	**90.4**	**80.9**	**94.1**	**61.4**	**40.1**	**89.7**	**72.6**
Source	30.4	3.5	34.4	46.7	44.6	0.6	37.9	29.3	48.3	28.0	78.9	33.0	28.9	34.5	14.1
Test	15.5	0.7	6.0	4.2	3.8	0.3	15.7	9.3	11.0	2.4	6.2	4.1	2.2	3.0	6.2
Admin	16.7	81.9	10.2	4.6	12.8	27.2	14.0	10.5	9.4	45.3	3.4	38.3	22.8	30.0	26.6
Combin.	37.4	13.0	49.4	44.5	38.9	72.0	32.4	50.9	31.2	24.3	11.4	24.6	46.0	32.5	53.1
Non-Linked	**36.0**	**1.7**	**39.4**	**55.6**	**51.4**	**15.9**	**10.0**	**4.8**	**9.6**	**19.1**	**5.9**	**38.6**	**59.9**	**10.3**	**27.4**
Source	14.4	2.3	17.8	35.3	7.1	6.0	13.6	1.9	41.4	21.9	39.7	12.0	39.8	45.3	18.7
Test	29.4	3.5	23.4	11.4	12.5	1.8	11.7	0.0	11.9	11.7	1.3	4.1	5.6	11.3	14.3
Admin	43.8	57.0	36.6	31.9	69.9	54.1	59.9	11.2	37.0	53.2	53.8	78.3	30.7	37.7	50.5
Combin.	12.3	37.2	22.2	21.4	10.5	38.1	14.8	86.9	9.7	13.2	5.1	5.6	23.9	5.7	16.5
Ratio	**0.6**	**0.0**	**0.7**	**1.3**	**1.1**	**0.2**	**0.1**	**0.1**	**0.1**	**0.2**	**0.1**	**0.6**	**1.5**	**0.1**	**0.4**
Source	0.5	0.7	0.5	0.8	0.2	10.0	0.4	0.1	0.9	0.8	0.5	0.4	1.4	1.3	1.3
Test	1.9	5.0	3.9	2.7	3.3	6.0	0.7	0.0	1.1	4.9	0.2	1.0	2.5	3.8	2.3
Admin	2.6	0.7	3.6	6.9	5.5	2.0	4.3	1.1	3.9	1.2	15.8	2.0	1.3	1.3	1.9
Combin.	0.3	2.9	0.4	0.5	0.3	0.5	0.5	1.7	0.3	0.5	0.4	0.2	0.5	0.2	0.3

Table 3.4: Share of commits (in %) categorized by the changed files and maximum values are gray colored.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
Linked	**39.4**	**87.2**	**62.3**	**61.3**	**36.2**	**24.4**	**33.5**	**43.9**	**64.5**	**52.5**	**36.6**	**62.5**	**45.2**	**54.1**	**39.8**
Bug	65.9	57.9	41.0	65.8	61.2	51.2	53.0	60.8	48.2	41.1	70.5	51.3	53.1	37.0	43.5
Feature	33.0	41.9	56.4	34.1	37.7	46.3	40.8	38.5	51.5	35.7	29.3	47.9	45.1	59.4	
Other	1.1	0.2	2.6	0.2	1.0	2.5	6.3	0.6	0.3	23.2	0.2	0.9	1.9	3.6	0.0
Non-Linked	**60.6**	**12.8**	**37.7**	**38.7**	**63.8**	**75.6**	**66.5**	**56.1**	**35.5**	**47.5**	**63.4**	**37.5**	**54.8**	**45.9**	**60.2**
Bug	69.2	51.3	33.6	62.9	59.6	49.7	43.4	54.6	35.5	55.7	75.2	43.2	43.9	23.6	61.7
Feature	29.7	47.4	65.6	36.0	40.2	47.8	52.1	43.6	63.2	36.5	24.2	55.5	53.1	67.9	38.3
Other	1.1	1.3	0.8	1.1	0.2	2.5	4.6	1.7	1.3	7.8	0.6	1.2	3.0	8.6	0.0
Ratio	**1.5**	**0.1**	**0.6**	**0.6**	**1.8**	**3.1**	**2.0**	**1.3**	**0.6**	**0.9**	**1.7**	**0.6**	**1.2**	**0.8**	**1.5**
Bug	1.1	0.9	0.8	1.0	1.0	1.0	0.8	0.9	0.7	1.4	1.1	0.8	0.8	0.6	
Feature	0.9	1.1	1.2	1.1	1.1	1.0	1.3	1.1	1.2	1.0	0.8	1.2	1.2	1.1	0.7
Other	1.0	6.5	0.3	5.5	0.2	1.0	0.7	2.8	4.3	0.3	3.0	1.3	1.6	2.4	N/A

Table 3.5: Share of resolved issues (in %) categorized by the changed type and maximum values are gray colored.

developers miss to report all the changes related to this issue. For example, the bug with issue key HIVE-6782[4] of the project Hive (P8) required multiple commits until it was resolved.

A closer look on the non-linked commits shows that the mean number of *test* and *admin* commits together build their largest share (57.3%). The ratio values of the commit data reflect that circumstance as well, because the highest ratios between the linked and non-linked values are in the categories *test* and *admin*. This can be an indicator that either no issue entries exist or that committers do not take care of referencing such kind of changes to an issue.

The survey data of the issues shows almost no difference in the distribution in the categories of linked and non-linked data. This is also reflected in the ratio values that indicated that the biggest deviation can be found in the *other* category. It is very likely that the reason for this variation can be found in the used issue workflow within a software project. For example, some projects introduce issue types in addition to the default set of types provided by the issue tracker.

Combined Commits

The commit overview in Table 3.3.2 shows that in six projects the highest number of linked commits is caused by so-called combined commits. In case of linked commits, most of the combined commits contain source code and test code changes (47.4%). Another share of combined commits (27.8%) consists out of a composition of source code, test code, and configuration changes. Followed by source code and configuration changes (18.4%), and test code and configuration changes (6.4%).

The combined commits of non-linked issues have a different constellation compared to the linked ones. The majority (52.2%) of the combined commits in this case contain source code, test code, and configuration changes. Commits consisting of source code and test code changes (26.2%) provide the second largest share, followed by source code and configuration changes (11.5%), and test code and configuration changes (10.1%).

[4] `https://issues.apache.org/jira/browse/HIVE-6782`

Interlinking Characteristics

During the survey, we also investigated interlinking characteristics in the analyzed Apache projects. We found two characteristics that occur in all of the projects. The first interlinking characteristic that we found reflects the situation of one commit that is associated to one issue and vice-versa. We call it *Loner*, because in case of a missing interlinking the according commit or issue is without any relationship. An example is the issue AMBARI-3412[5] of Ambari (P2) and the associated, but not linked commit f2146a41[6]. The problem is that the commit message contains the issue key in the wrong format and therefore the link is not detected. We call the second interlinking characteristic *Phantom*. In case of this characteristic, an issue has multiple associated linked commits, but there are further commits that are associated with, but not linked to the issue. These associated, but not linked commits are called Phantoms, because the link is not explicitly established and therefore not visible. An example for a Phantom is CAMEL-7354[7] and the not linked commit 14cd8d6b[8]. These characteristics represent an initial set of interlinking characteristics that we found in the analyzed projects. Software projects can contain further characteristics as well and the PaLiMod model can be extended to cover these additional cases. An example scenario is a commit that addresses multiple issues at once, but references only one of them. Thus, the existing linking is correct, but since the commit addresses multiple issues, the linking does not tell the whole truth. Such a characteristic is a candidate for extension and a potential subject for future work.

[5] https://issues.apache.org/jira/browse/AMBARI-3412

[6] http://git-wip-us.apache.org/repos/asf/ambari/commit/f2146a41

[7] https://issues.apache.org/jira/browse/CAMEL-7354

[8] http://git-wip-us.apache.org/repos/asf/camel/commit/14cd8d6b

3.4 Discovering Loners and Phantoms

The investigation on the Apache survey data showed that linked commit and issue data follow certain characteristics. We examined the characteristics of Loners and Phantoms, and propose heuristic approaches to automatically interlink commit and issue data with these characteristics. Loners and Phantoms can be quantified by a retrospective analysis of the data in the VCS and the issue tracking platform.

3.4.1 The Loner Heuristic

The aim of this heuristic is to discover issues which get fixed by a single commit. It tries to detect candidates based on a combination of time and committer information and discards wrong candidates using commit and reopen history. The heuristic is specified as follows:

$$\begin{aligned}\forall x, \forall y, \exists z \Big(& \big(ResolvedCR(x) \wedge NonLinkedCommit(y) \\ & \wedge CRResolvedEvent(z) \\ & \wedge HasCRElement(x, z) \\ & \wedge NonLinked \wedge PersonCond \\ & \wedge TimeCond \wedge ReopenCond \\ & \wedge CommitCond\big) \implies LinkCandidate(x, y)\Big)\end{aligned}$$

$$NonLinked := \neg\exists w\big(Commit(w) \wedge isExplicitlyLinked(x, w)\big)$$

The results of Loner heuristic are pairs (x, y) of resolved change requests x and not linked change sets y, if all conditions, i.e. *PersonCond*, *TimeCond*, *ReopenCond*, and *CommitCond*, are satisfied.

$$PersonCond := hasResolved(x) = isCommittedBy(y)$$

The *person* condition (*PersonCond*) requires that the issue resolver, i.e. the person who resolved the change request x, is the same person as the change set committer, i.e. the person who committed the change set y.

$$\begin{aligned} TimeCond :=& resolvedTS(x) - commitTS(y) < ParTime \wedge \\ & resolvedTS(x) - commitTS(y) > 0 \end{aligned}$$

The *time* condition (*TimeCond*) specifies the maximum time-span between a commit and an issue resolution and is based on the parameter *ParTime*. We use a time frame of five minutes for the parameter as the analysis of the 15 projects has shown that 55.4% of the issues in the 1:1 case are resolved within 5 minutes after the commit.

$$\begin{aligned} CommitCond :=& \neg\exists v \big(Commit(v) \wedge \\ & isCommittedBy(v) = isCommitted(y) \wedge \\ & v \neq y \wedge commitTS(v) > commitTS(y) \wedge \\ & commitTS(v) < resolvedTS(x) \big) \end{aligned}$$

The *commit* condition (*CommitCond*) checks that there are no commits addressing other issues between a commit and an issue linking candidate. Thus, there shall be no other commit v by the committer of y which lies between the commit time of y and the resolve time of x. The condition is required, because otherwise no exact matching of commits to issues would be possible. Imagine a second not linked commit within the considered time frame, the issue could then also have multiple associated commits (*Phantoms*).

$$\begin{aligned} ReopenCond :=& \neg\exists u \big(CRResolvedEvent(u) \\ & \wedge hasCRElement(x, u) \wedge u \neq z \big) \end{aligned}$$

The *reopen* condition (*ReopenCond*) ensures that reopened issues are not taken into account. It does this by checking the non-existence of an arbitrary resolved event u of the change request x, which has to be different than z. If

an issue has multiple resolve-events, i.e. it has been reopened at least once, it contains very probably further patches and thus, the issue would belong to the 1:n scenario again and has to cope with *Phantom* commits.

The remaining term in the heuristic's specification, *NonLinked*, guarantees that the resolved change request has no associated commits and is therefore not linked. It achieves this by checking for non-existence of a commit linked to the change request. The first four predicates of the heuristic ensure that a right set of candidates is chosen, i.e. a resolved change request x, which has an associated change request resolved event z, and a non-linked commit y. If this set of candidates satisfies all conditions a *LinkCandidate* is found.

The used functions and predicates are derived by the annotations of the model. For example, the function *isCommittedBy(y)* can be seen as a getter returning the person which has contributed the commit y. Similarly, the predicate *HasCRElement(x,z)* checks, whether there is the relation *hasCRElement* between the change request x and the change request resolved event z. Predicates such as *NonLinkedCommit(y)* simply check for a specific class membership, in the given example if y is a non-linked commit. The function *commitTS* returns the point in time of the commit. Similarly, *resolvedTS* returns the timestamp of the issue-resolution. The comparison of identities is achieved by comparing names and email addresses.

3.4.2 The Phantom Heuristic

The aim of this heuristic is to discover non-linked commits (Phantoms) that address issues with at least one associated linked commit. An existing link between a commit and a change request serves as the baseline of the approach. It detects potential commits that address the same change request based on the commit time and the touched resources. The heuristic is specified as follows:

$$\forall x, \forall y \Big(\big(NonLinkedCommit(x) \wedge LinkedCommit(y) \wedge \\ PersonCond \wedge TimeCond \wedge ResCond \big) \\ \implies LinkCandidate(x, y) \Big)$$

The results of the Phantom heuristic are pairs (x, y) of linked commits x and non-linked commits y, if all conditions, i.e. *person*, *time*, and *resource*, are satisfied.

$$PersonCond := isCommittedBy(x) = isCommittedBy(y)$$

The *person* condition (*PersonCond*) requires that the committer of a non-linked candidate x has to be the same person as the committer of the baseline commit y.

$$TimeCond := abs(commitTS(x) - commitTS(y)) < ParTime$$

The *time* condition (*TimeCond*) ensures that the maximum time-span between a non-linked candidate commit and the baseline commit is *ParTime* days, being *ParTime* a specifiable parameter. Typically, more complex issues are not solved using single commits, thus multiple code fragments are committed, usually scattered across one or more work days. In order to take such behavior into account, the time frame for the Phantom heuristic, i.e. the value of *ParTime*, is set to 5 days, which is small enough to exclude unrelated commits and big enough to include potentially interesting commits.

$$ResCond := similarity(hasResource(x), hasResource(y)) \\ > ResOverlap$$

Finally, the *resource* condition (*ResCond*) requires that the lists of changed resources of two commits have to overlap of at least *ResOverlap* percent. The

analysis of existing links in this 1:n scenario has shown that there is an overlapping of 65% of the commits' associated resource files. For this reason we have set the parameter *ResOverlap* for resource coverage to 66%.

The predicates *NonLinkedCommit(x)* and *LinkedCommit(y)* ensure that the heuristic is applied on pairs of non-linked commits and linked commits. If all conditions match, a *LinkCandidate* is found. Similar to the Loner heuristic, functions and predicates are highly related to the model's subjects and annotations. Additionally, the function *abs* is used, which returns the absolute value of the given number. It is required to ensure that a commit lies within the considered time frame. The function *similarity* takes two lists of file names and calculates the resource coverage by dividing the number of occurrences of the larger list's elements in the smaller one by the total number of elements in the larger list. File names are provided by the function *hasResource*, which takes a change set as argument and returns a list of its associated file names.

3.5 Evaluation

A central aim of our approach is to reduce the residual of non-linked commit and issue data in software projects, which have interlinking guidelines established. For the evaluation, we used a scenario-based approach to simulate different residuals of non-linked commit and issue data. Such a simulation was necessary because to the best of our knowledge no baseline dataset is publicly available for partial interlinked software projects. For each scenario, we randomly removed a percentage (10%, 20%, 30%, and 40%) of existing links from a project's dataset. The resulting set of commit and issue data built the input for the evaluation runs and the original dataset built the baseline used for the precision and recall calculation.

The random removal of links was done automatically and took care of certain constraints to avoid impossible interlinking constellations in the resulting datasets. For example, a random removal without constraints in case of a Phantom might unlink all commits of an issue, which results in a constellation that no longer fits the definition of a Phantom. For the random link removal

we applied different combinations of three constraints that are based on the findings of our project survey (see Section 3.3.2). The first constraint is the *link* constraint to ensure that at least one link to an issue with multiple links remains in case of a Phantom scenario. The *time* constraint is a constraint that shall avoid cases in which multiple, but time independent links to one issue get removed. For example, an issue gets re-opened after half a year. Such a situation leads to additional commits that address the same issue but are time independent. The third constraint, i.e. *person* constraint, ensures that the issue's resolver also committed the patch (Loner heuristic), or that the committer of the linked commit is the same as the committer of the not linked commit (Phantom heuristic). This constraint is especially important for scenarios with Loners, because of the limited amount of attributes that are available to detect and confine them from other scenarios, such as the Phantom.

We ran the evaluation for each heuristic and each of the different scenarios ten times to mitigate the risk of potential outliers introduced by the random manner of the deletion process and the used constraints. A Java application was used for the automatic execution of the evaluation runs and the collection of the respective results.

Overall, the evaluation results showed that both heuristics achieve high precision and recall. Unless otherwise stated, the values presented here were extracted from the evaluation runs based on 30% removed existing links. This rate is based on the average percentage of non-linked commits which is about 26% as presented in Section 3.3.2. In case of the Loner heuristic, the overall precision is 96% with a recall of 92%. In case of the less restrictive *time* constraint, the precision remains stable and the recall decreases to 70%. The F-measure reduces from 0.94 to 0.81. For the Phantom heuristic, the overall precision is 73% with a recall of 53%. Moving from the combined *time-link* constraint to the single *link* constraint leads to a precision of 68% and a recall of 39%. The F-measure decreases from 0.61 to 0.50.

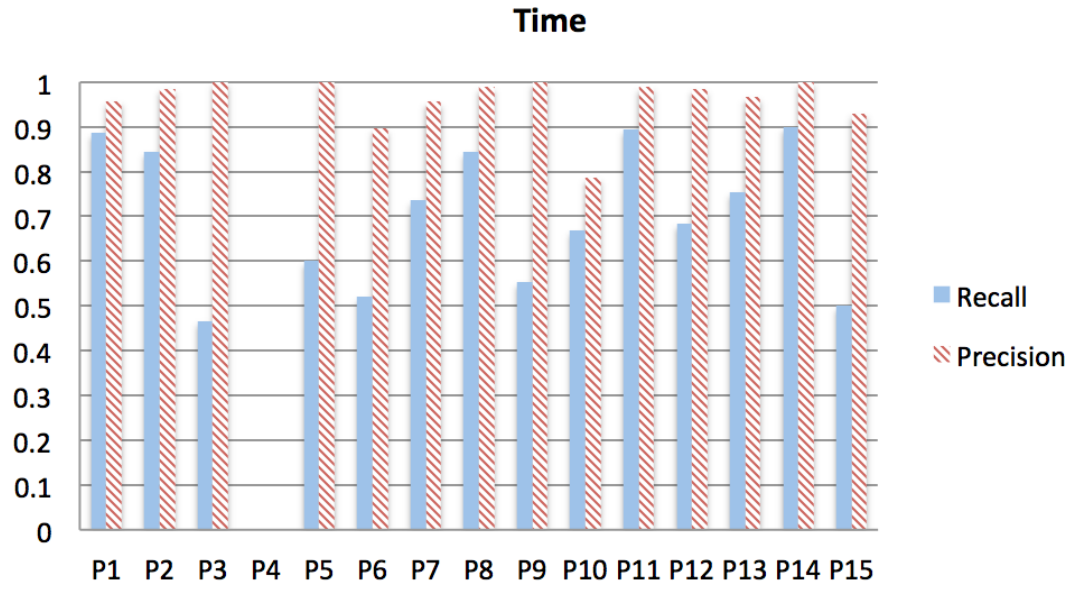

Figure 3.3: Loner: *time* constraint

3.5.1 Loner Heuristic

Figure 3.3 illustrates the average results of the evaluation of the Loner heuristic after 10 runs under the *time* constraint. The heuristic achieves in 5 projects precision and recall values beyond 80%. At a first glance, CXF (P4) seems to be an outlier where the Loner heuristic does not work at all. The bad performance of the Loner heuristic in case of CXF is caused by a too small set of one-to-one mappings between commits and issues. There exist 25 links suited as candidates for deletion (unlinking), but when applying the *time* constraint this number shrinks to only three remaining candidates. Therefore our application for pursuing the evaluation did not remove any links as even a single link representing 33% of the delete candidates exceeds the specified threshold of 30%. In the 40% removal setting, in each run one link is deleted and in 80% of the cases correctly reestablished by our heuristic. Another interesting project is Camel (P3), which has a high precision and a rather low recall value under the *time* constraint. After a manual analysis of the data we found that one project member is responsible for approximately 45% of the commits. Due to the usage of different email addresses by this developer

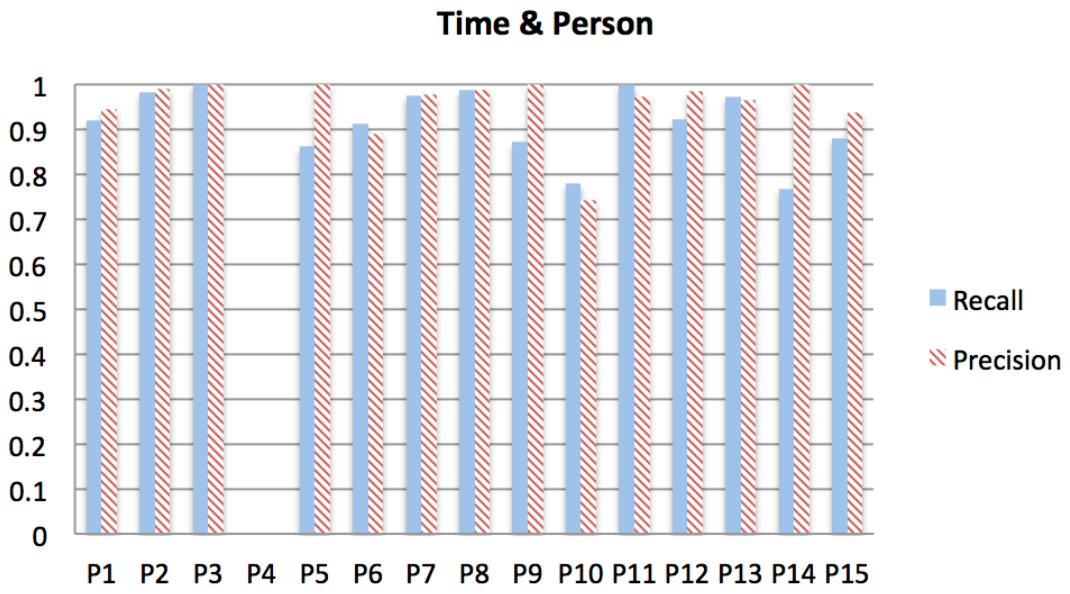

Figure 3.4: Loner: combined *time-person* constraint

on the two platforms that were used to fill our model with data, i.e. version control and issue tracker, our heuristic's person condition (based on name and email information) cannot establish a matching between these identities and therefore almost every second deleted link cannot be re-discovered for this project. As previous work has shown, the problem of identity matching is more than an isolated case and happens quite frequently in Apache Software Foundation projects and others [Bird et al., 2006]. The Loner heuristic does not make use of existing links, therefore the results vary only slightly in the different test runs (10%, 20%, 30%, and 40% deleted links), involving the precision values being nearly constant and a variation of about 2% regarding recall. Given the application of the combined *time-person* constraint in the Loner heuristic (see Figure 3.4), increased recall values can be achieved which is based on the fact that this variant strongly covers the heuristic's conditions.

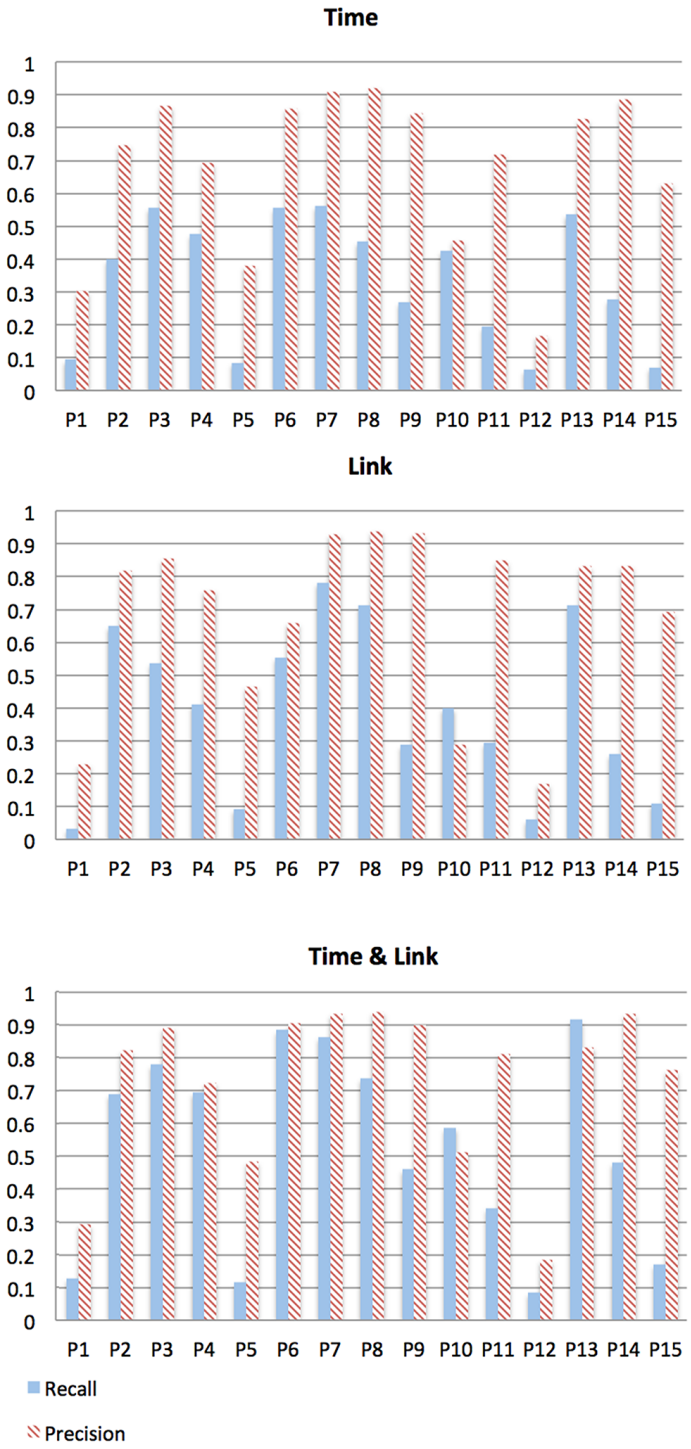

Figure 3.5: Phantom: time, link, and time-link constraints

3.5.2 Phantom Heuristic

As depicted in Figure 3.5, the results of the Phantom heuristic are subject to higher fluctuations. For some projects high precision and recall values are achieved, while for a few projects both precision and recall values are rather low. Even the combination of time and link constraints does not lead to a significant improvement for certain projects. In general, we observe that the combined variant leads to smaller gaps between precision and recall values as can be clearly seen between the time, link, and time/link charts for the Phantom heuristic in Figure 3.5.

The Phantom heuristic uses existing links to identify candidates. Thus, the more links are removed during our deletion process, the smaller the base of existing links to learn from is, for example for determining the source files which are frequently updated. This fact is reflected in the decrease of the recall value which drops from 47% in the 10% setting to 27% when removing 40% of the links. The *link* constraint in our simulation mitigates this behavior and reduces this effect from 20% down to 4%.

We conducted further experiments with the Phantom heuristic to better understand and improve the results. Therefore, we tested different parameter settings for the heuristic's conditions, strengthened, weakened or removed some of them in order to see how the results evolve. First, we adjusted the person condition such that we require the committer of the non-linked candidate to be part of the issue's discussion by contributing at least one comment. Under the *link* constraint the modified heuristic detected less candidates, the precision improved by 4%, but the recall dropped by 10% on average. Second, we removed the person condition completely. Again, we used the *link* constraint in the 30% setting to compare the results. The number of identified candidates increased by 21%, the precision decreased by 5% and the recall increased about 3%. In a next step we combined this setting with a time-span of the heuristic's time condition broadened from 5 to 7 days, thus setting the parameter *ParTime* to 7. Recall values remained unchanged, but precision values decreased by further 2%. In total we suffered the loss of 7% precision and gained just 3% in recall. The F-measure remained stable at 0.50.

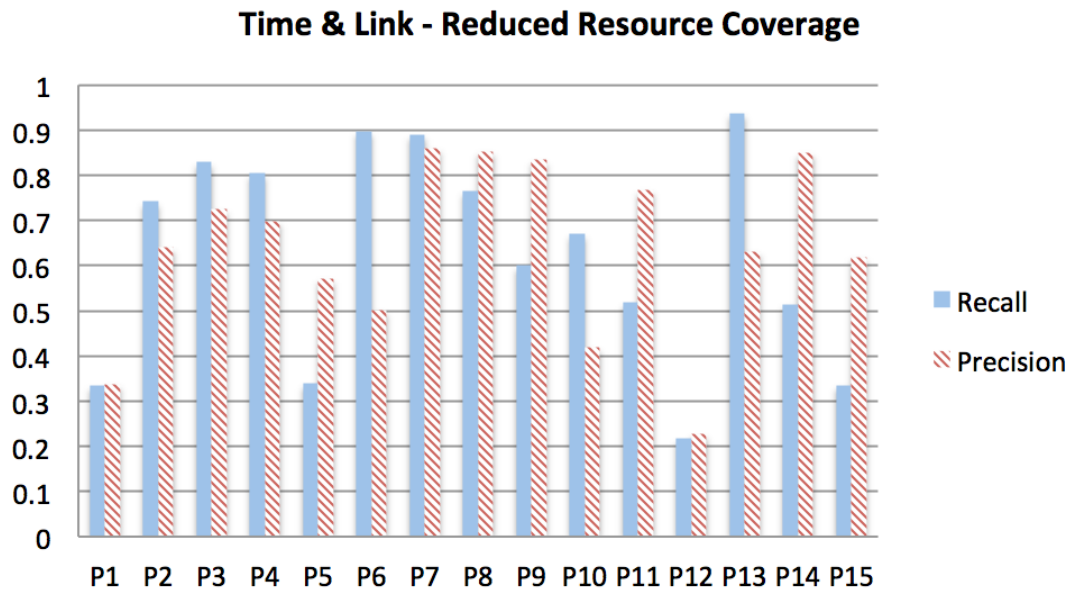

Figure 3.6: Phantom: time-link constraint, resource coverage set to 30%

Finally, we tackled the remaining resource condition as well. We set the rate to which degree changed resources of two commits have to overlap, i.e. the *ResOverlap* parameter, from 66% to 30%. Under the *link* constraint we increased recall values by 8% on average and determined a decrease of 10% in precision. With a combined *time-link* constraint we observed an increase of 10% regarding recall and a drop of 9% in precision, so we ended up with 64% precision and 63% recall. The F-measure improved by 0.02 to 0.63. The results for the single projects are depicted in Figure 3.6.

3.6 Discussion

Overall, the evaluation results showed that our proposed model and the proposed heuristics can be used to re-establish missing links in partly interlinked commit and issue data. We discuss the implications of the survey findings, the proposed heuristics and their performance, and the potential of the interlinking model.

3.6.1 The Partial Linking Model

The results of our evaluation showed that the PaLiMod model enables a fast integration of data from different data sources (e.g., VCS, issue tracking platform). The graph-based structure of the model provides a powerful mechanism to describe relationships and attributes of data entries independent of the data source. Currently, PaLiMod interlinks data from VCSs and issue tracking platforms, but it is possible to include discussions from mailing lists, chat protocols or other project related information. A further benefit of the PaLiMod model is data abstraction. For example, the definition of analysis algorithms or heuristics on top of the PaLiMod model can take place without knowledge about the actual structure of the data in the origin data source. In this paper, we evaluated the performance of the proposed heuristics with projects that use Git and JIRA, but the heuristics would also work for data integrated from SVN or Bugzilla.

However, one limitation of the PaLiMod model, similar to other model-based approaches, is the transfer of new data (e.g., newly established links) back to the origin data source (e.g. VCS). We showed in our evaluation that the proposed heuristics and the PaLiMod model can be used to re-establish missing links, which can be exported into a data file or a report. As far as we know, there is no standardized way to integrate such heuristically established links into an existing issue tracking platform or into the history of a VCS.

3.6.2 Commit and Issue Interlinking

From the perspective of mining software repositories, developers process a stack of issues in a sequential order and every issue gets fixed by an explicit commit. However, the survey showed that such an ideal case is hard to find in practice. Patches and new features are contributed in arbitrary order and single-patch-fixes and other contributions tend to overlap.

Nevertheless, we were able to extract a set of interlinking characteristics used in the analyzed projects. We called them *Loner* and *Phantom*. A special challenge is the distinction of such characteristics in order to be strict enough

to isolate from each other without refusing eligible candidates at the same time. Between the *Loner* and the *Phantom* the distinction of the characteristics is challenging especially in cases where two or more issues get fixed in concurrent. For example, commit C1 addresses issue I1 and commit C3 addresses issue I1. In between these two commits, the same developer commits another source change C2 without an issue key and changes the status of issue I2 to resolved. In such a case, the overlapping of the changed resources of the single commits can be used to determine the belonging of C2. Despite the information about the overlapping it is not possible to deterministically compute the belonging of such a commit.

3.6.3 Interlinking Heuristics

Both of the proposed interlinking heuristics perform well in most of the cases, however there are projects in which they don't because of various, often project specific, factors. The primary attribute used for all of the heuristics is the temporal relation of two events. For example, if Loners are within the specified time window there is a high chance that the heuristic will detect them and create a correct linking. Another important attribute of our approach is the matching of developer identities between identities used in the VCS and identities used in the issue tracking platform.

Moreover, the heuristics are based on the assumption that the source code contribution originates from the same person which changes the issue status to resolved. However, this assumption does not always hold and depends on the specific project. Especially the evolution of the recall values between Figures 3.3 and 3.4 reflects the influence of this assumptions onto the overall results. For example, for the project ActiveMQ (P1) the difference is small and our condition applies very well. In contrast, Hadoop (P6) has a divergence in its recall values which is, from our perspective, caused by code contributions and issue resolutions that are carried out by different persons.

During the experiments with various attributes and their settings, we identified the resource overlapping attribute and its threshold (*ResOverlap*)

as the major impact factor on the Phantom heuristic. The threshold was determined empirically across all of the analyzed projects, which leads to better results for projects that have a low standard deviation. Our experiments have shown that the higher the overlapping of changed resources between the commits in a project are, the higher the recall. The downside of reducing the threshold is that for the projects with high overlapping, the precision drops sharply. For example, this can be recognized in projects Hadoop (P6) and Spark (P13) in which the precision decreases up to 40%. Nonetheless, there are projects, especially ActiveMQ (P1) and Sling (P12) where the Phantom heuristic does not perform well. The reasons are manifold and the mentioned resource condition is just one factor. Other factors are the general project organization, e.g. the number of active contributors, or even on the source code level, when multiple issues are fixed within a few days addressing only a small amount of resource files. In such cases, our heuristic tends to produce a lot of false positives, for example in the Sling project (P12).

It is important to highlight, that in difference to other research in this area, the proposed heuristics are very simple as they don't use rather complex techniques such as text-similarity to discover links. Our heuristics are based on simple information including developers, commits and issues which represent the essential elements in VCSs and issue tracking platforms. For example, they don't rely on artifacts like comments which might exist for certain issues, but usually not for all of them. Therefore, our heuristics are applicable to a broader set of cases, and despite of their simplicity, good results have been achieved.

3.7 Threats to Validity

Empirical studies have limitations that have to be considered when interpreting their results. Our study is amenable to threats to the external and internal validity.

External Validity. For the extraction of the commit and issue interlinking, we relied on data gathered by mining source code repositories and issue tracking platforms of 15 different Apache projects. We limited the data extraction to

a period of one year. These decision might limit the generalizability of our results, and further studies might need to be conducted to verify that our results can also be applied to other projects. However, to mitigate this risk, we have chosen the projects used in our analysis in a way to get a broad sample of various projects with different characteristics. Another threat that might limit the generalizability of our results is the use of only one type of VCS (Git) and only one type of issue tracking platform (JIRA). A different scope of functionality provided by a VCS or an issue tracking platform can lead to a different interlinking behavior.

Internal Validity. We analyzed interlinking characteristics of 15 Apache software projects and derived a set of interlinking heuristics based on these findings. Based on this approach, we were able to extract the most prominent interlinking characteristics used in projects. There might be further interlinking characteristics, which only occur in single projects of our dataset. We tried to mitigate this by ensuring a large coverage of the links found in our dataset by the proposed heuristics.

The dataset used for the evaluation of the interlinking heuristics was derived from the original dataset used in the survey. This reuse of the dataset may influence the performance measurements of the heuristics during the evaluation. One possibility to overcome this threat can be the use of a dataset, in which the residual of non-linked commit and issue data was manually annotated by an active member of the according project. Due to the absence of such a manual annotated dataset, we minimized this threat by a random removal of links from interlinked commit and issue data. Furthermore, we repeated each evaluation ten times to mitigate a bias introduced by the random removal.

3.8 Related Work

The interlinking of development artifacts (e.g., commits, issues, emails) has been addressed multiple times in related literature. In the following, we discuss the most relevant related work in the area of commit and issue interlinking.

Mockus and Votta [Mockus and Votta, 2000] introduced an early approach to extract a textual description of the purpose of a change set from a VCS only. One of the first approaches that uses data from a VCS in combination with data from an issue tracker was presented by Fischer et al. [Fischer et al., 2003b]. The incentive of their work was the research on software evolution, which relies on detailed and rich data of software projects.

Śliwerski et al. [Śliwerski et al., 2005] introduced an approach to automatically detect changes that fix a certain bug. Their approach is built on top of an interlinked commit and issue dataset. Kim et al. [Kim et al., 2006] proposed an approach to automatically detect bug-introducing patterns based on data mined from fixed bugs. They interlinked commit and issue data for the change pattern extraction.

Ayari et al. [Ayari et al., 2007] addressed threats on building models based on commit and issue data. They showed that a considerable number of links cannot be established based on a numerical issue id in the commit message. Bird et al. [Bird et al., 2009] investigated the impact of bias in commit and issue datasets used in research, such as in the bug prediction area. Their experiments showed that bias indeed influences the performance of bug prediction approaches, such as BugCache [Kim et al., 2007]. One proposed way to overcome this bias is the use of explicit linking mechanisms as offered by platforms, such as *Rational Team Concert*.[9] Bachmann and Bernstein [Bachmann and Bernstein, 2009] proposed a set of software engineering process measurements based on commit and issue data. A survey of five open source projects and one closed source project showed major differences in the engineering process expressed by the commit and issue data. In further work [Bachmann et al., 2010], Bachmann et al. established a ground truth of commit and issue data from the Apache Httpd project created by a core developer of the project. Their analysis showed that bugs get often fixed without an issue entry and that commits not always change functionality of a software system. They introduced a tool call Linkster [Bird et al., 2010], which provides capabilities for manual commit and issue interlinking. A series of approaches [Wu et al.,

[9] https://jazz.net/products/rational-team-concert/

2011, Nguyen et al., 2012, Bissyande et al., 2013] tried to overcome the raised restrictions (e.g., [Kim et al., 2007, Bachmann and Bernstein, 2009, Bachmann et al., 2010]) of data interlinking based on numerical issue ids only and proposed interlinking heuristics based on, for example, text similarity (e.g., [Wu et al., 2011]) or change set overlapping (e.g., [Nguyen et al., 2012]).

There are other research areas that address the interlinking of source code artifacts other than commit and issues. For example, Bachelli et al. [Bacchelli et al., 2009, Bacchelli et al., 2010] investigated the interlinking of email data from mailing lists with commit or patch data. The area of traceability research (e.g., [Antoniol et al., 2000, Corley et al., 2011]) addresses the interlinking of source code changes with documentation changes.

Our approach differs from the mentioned related work, as we address the interlinking of commit and issue data in environments that use already interlinking, but have a residual of non-linked commit and issue data. The unused information stored in such a residual can foster approaches (e.g., bug triaging), which heavily rely on the mined data of VCS and issue tracking platforms. To best of our knowledge, such a scenario is not covered by any existing work.

3.9 Conclusion

The interlinking of commit and issue data has become a de-facto standard in software projects, which is reflected in guidelines of large open-source communities [Apache Software Foundation,]. In this work, we (1) investigated the extent to which developers follow such interlinking guidelines by a survey of 15 Apache projects, (2) deducted a set of interlinking characteristics found in the surveyed projects, and (3) proposed a model (ParLiMod) and a set of heuristics to interlink the residual of non-linked commits and issues.

We observed that in the majority of the analyzed projects, the number of commits linked to issues is higher than the number of commits without link. On average, 74% of commits are linked to issues and 50% of the issues have associated commits. Based on the survey data, we identified two interlinking

characteristics which we call *Loners* (one commit, one issue) and *Phantoms* (multiple commits, one issue). For these two characteristics, we proposed heuristics to automatically interlink non-linked commit and issue data. The evaluation results showed that our approach can achieve an overall precision of 96% with a recall of 92% in case of the Loner heuristic and an overall precision of 73% with a recall of 53% in case of the Phantom heuristic.

Potential future work includes the analysis of further software projects to allow for a quantitative description of the additional characteristic (one commit addresses multiple issues) we have discovered, but have not tackled yet. The model provides also a solid basis for the development of more complex heuristics, including e.g., text-similarity.

4

SQA-Profiles: Rule-Based Activity Profiles for Continuous Integration Environments

Martin Brandtner, *Sebastian C. Müller, Philipp Leitner, and Harald C. Gall*
Published at the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, 2015
DOI: 10.1109/SANER.2015.7081840

Abstract

Continuous Integration (CI) environments cope with the repeated integration of source code changes and provide rapid feedback about the status of a soft-

ware project. However, as the integration cycles become shorter, the amount of data increases, and the effort to find information in CI environments becomes substantial. In modern CI environments, the selection of measurements (e.g., build status, quality metrics) listed in a dashboard does only change with the intervention of a stakeholder (e.g., a project manager). In this paper, we want to address the shortcoming of static views with so-called Software Quality Assessment (SQA) profiles. *SQA-Profiles* are defined as rule-sets and enable a dynamic composition of CI dashboards based on stakeholder activities in tools of a CI environment (e.g., version control system). We present a set of SQA-Profiles for project management committee (PMC) members: Bandleader, Integrator, Gatekeeper, and Onlooker. For this, we mined the commit and issue management activities of PMC members from 20 Apache projects. We implemented a framework to evaluate the performance of our rule-based SQA-Profiles in comparison to a machine learning approach. The results showed that project-independent SQA-Profiles can be used to automatically extract the profiles of PMC members with a precision of 0.92 and a recall of 0.78.

4.1 Introduction

Software development has become a data-driven discipline [Buse and Zimmermann, 2012] and the tools used for Continuous Integration (CI) are important data sources in the development life cycle. The way of accessing data from CI environments differs between the stakeholders of a software project. For example, developers primarily perceive the CI-process in case of build exceptions (e.g., build or test failure), whereas software managers actively consolidate CI environments to gather data for planning and decision making purposes. The term *CI environment* in the context of our work refers to all platforms that are involved to perform and manage the automatic integration of source code changes. Such an environment typically consists of a version control system (VCS) and a issue tracking platform as well as other tools.

In earlier work [Brandtner et al., 2014, Brandtner et al., 2015a], we introduced a data integration approach for CI-data called SQA-Mashup. Our study

showed that the proposed role-based tailoring fosters the interpretation of CI-data in a fast and accurate way. However, the composition and tailoring of the different views in state-of-the-art CI-tools as well as in SQA-Mashup is rather time-consuming and needs to be done by a professional. We propose activity data mining to overcome this shortcoming for enabling a fully-automatic composition of views, and a tailoring of CI-data according to the activities of a stakeholder. The use of the mined activity data is not restricted to visualization of CI-data. Additionally, it can also be used for project management purposes, such as workload reporting.

In this work, we propose a rule-based approach to automatically profile stakeholders based on their activities in the version control system (VCS) and the issue tracking platform to enable the tailoring of data generated by CI-tools that operate on top of the VCS and issue tracking platform. We introduce so-called *SQA-Profiles* to describe the characteristic activity patterns of stakeholders within a certain role. For example, the project management committees (PMCs) of Apache projects are groups of contributors, who lead the project's development and community. The size of PMCs varies between 9 (Apache Jena) and 55 (Apache Httpd) members.[1] Despite all PMC members having the same formal roles, the actual task focus varies substantially between stakeholders. For example, one PMC member might take care of patch integration, while another handles issue management.

The aim of our approach is the establishment of a model for a project-independent definition of stakeholder profiles based on activity data. We analyze the activities of PMC members from 20 Apache projects, and derive a set of SQA-Profiles for PMCs. We extract the last year's project histories from the VCS and the issue tracking platform, and use a k-means clustering to categorize the activity data, and to derive rules for the definition of SQA-Profiles based on the characteristics of each resulting cluster. Additionally, we introduce a nominal scale of activity data with the values *High*, *Medium*, and *Low* to enable a project-independent and human-readable rule definition. For example, the cluster with a high merge activity and a medium or high commit

[1] `http://people.apache.org/committers-by-project.html`

activity forms the foundation for a profile describing the work of a stakeholder that integrates patches. The resulting set of SQA-Profiles covers *Bandleaders*, *Integrators*, *Gatekeepers*, and *Onlookers*.

The main contributions of this paper are as follows:

- **A model to describe activity profiles of stakeholders in a project-independent manner.**
- **A set of project-independent PMC member activity profiles**.
- **A framework to automatically profile stakeholders based on activity data mined from the VCS and the issue tracking platform.**

We implemented a prototypical framework (*SQA-Profiler*) to evaluate the performance of our rule-based approach. In this evaluation, we investigated whether a rule-based and project-independent approach, such as the one used by SQA-Profiles, can achieve a similar performance as a machine learning based approach, which has to be individually parameterized for each project. The results show that our approach can indeed determine profiles of PMC members with a precision of 0.92 and a recall of 0.78.

The remainder of this paper is structured as follows. In Section 4.2 we present the methodology we followed in this paper. In Section 4.3, we introduce our rule-based approach and a set of SQA-Profiles, followed by the evaluation in Section 4.4. We discuss results in Section 4.5, and threats to the validity of our research in Section 4.6. The most relevant related work is discussed in Section 4.7. Finally, we conclude with our main findings in Section 4.8.

4.2 Approach

The aim of our approach is the profiling of stakeholders within a PMC. In Apache projects, the membership in a PMC is treated as role.[2] In comparison to other roles, such as contributor, the covered spectrum of tasks is broader

[2] `http://www.apache.org/foundation/how-it-works.html`

in a PMC. Project committers work on issues and contribute source code changes. PMC members actively contribute issues and source code as well, but the PMC is additionally in charge of project and community management. The management of the project incorporates tasks, such as monitoring or gatekeeping. A benefit of using a committee compared to a single manager is the ability to share tasks among the different committee members. However, the resulting different focus of the PMC members requires a different view on the data presented in dashboards as well [Buse and Zimmermann, 2012].

The extracted profiles can be used for an automatic composition of views or for a tailoring of CI-data in accordance to the activities of a PMC member. We address the goal of our approach with the following research questions:

RQ1: *Can activity data mined from the version control system and issue tracking platform be used for the extraction of profiles within a PMC?*

RQ2: *What profiles of PMC members can be extracted from the activity data, and how can these profiles be described in a ruled-based model?*

To answer these research questions, we studied the activity data of PMC members from 20 Apache projects between September 2013 and September 2014. All selected projects are Java projects that use Maven as build tool. We decided to analyze the time-range of one year instead of the entire project history to minimize the noise introduced by PMC member changes. The extracted activity data include the project name, the stakeholder associated with the activity, and the number of each of the following events: commits, merges, issue status changes, issue comments, issue assignee changes, and issue priority changes. These events are referred to as *attributes* in the remainder of the paper. In total, we ended up with 8'707 data points extracted from the VCS[3] and the issue tracking platform[4] of the according projects.
Figure 4.1 depicts the five phases of our approach:
Phase 1 - Data extraction
Extracting activity data from the VCS and the issue tracking platform.
Result: Stakeholder activity data records.

[3] `https://github.com/apache`
[4] `https://issues.apache.org/jira/`

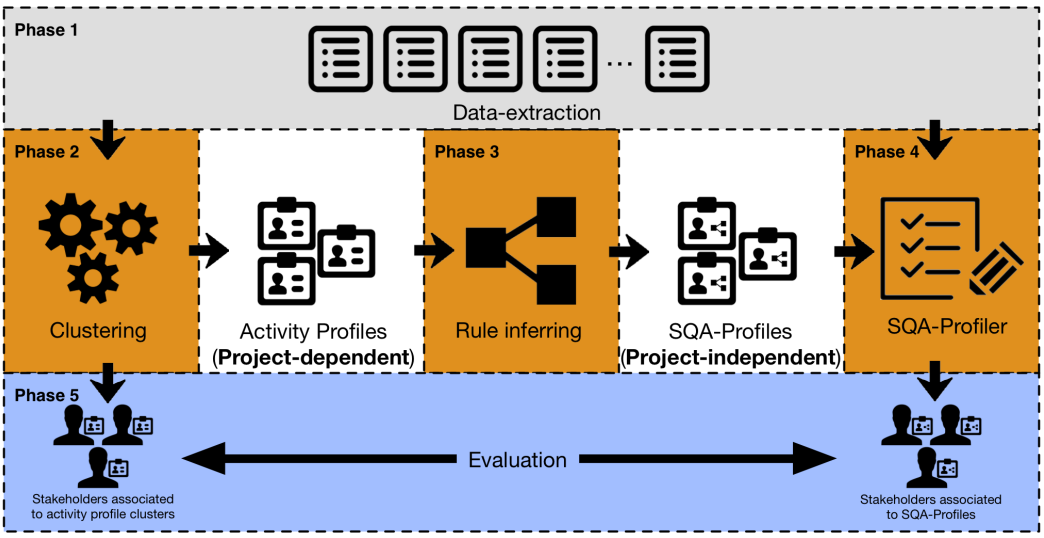

Figure 4.1: Overview about the five phases of our approach

Phase 2 - Clustering
Clustering the extracted activity attributes.
Result: Stakeholders clustered based on their activity data. The clusters were computed based on the numerical activity values.
Remark: These values are **project-dependent** and can slightly vary between different projects.

Phase 3 - Rule inferring
Inferring project-independent activity profiles from the clusters generated in Phase 2. A nominal scale is used as abstraction layer to introduce project-independent values for the rule definition.
Result: A set of project-independent and human-readable activity profiles, called *SQA-Profiles*.
Remark: These values are **project-independent**.

Phase 4 - SQA-Profiler
Executing *SQA-Profiler*, based on the extracted activity data (Phase 1) and the derived SQA-Profiles (Phase 3).
Result: Associations of stakeholders to a SQA-Profile.

Phase 5 - Evaluation
Evaluating the results of the profile association process of Phase 2 and Phase 4. The two association processes are different as in Phase 2 the associations are computed based on project-dependent activity profiles, and in Phase 4 the associations are computed based on project-independent activity profiles.
Result: The performance of the proposed SQA-Profiles approach compared to a project-dependent baseline.

The aim of Phase 3 is to abstract activity profiles into rules to mitigate the discussion of thresholds needed for the definition of activity profiles. For example, it is hard to define a numerical and project-independent threshold for a "high" commit activity. Therefore, we use a nominal scale computed by machine learning to define high, medium, and low values for each project individually. This adds a layer of abstraction, which enables a project-independent and human-readable definition of activity profiles.

In the evaluation, we compare the stakeholder profile associations computed by our rule-based approach against a baseline computed by machine learning. We decided to use a ground truth computed with machine learning instead of survey data, as the survey data might not be as objective as the values in the repository. We are aware that not all roles must be visible in the activity data of VCS and issue tracking platforms (see [Aranda and Venolia, 2009]). However, what we want to address with our approach is the circumstance that the self-described role or profile of a stakeholder can deviate from the actual role or profile.

4.3 Activity Profiling

Next, we describe each phase of our profiling approach.

4.3.1 Phase 1 & 2 - Data Extraction and Clustering

We merged the activity data extracted from the VCS and the issue tracking platform based on the user accounts, and combined those activities that are

associated with the same email address. However, we noticed that some developers used different email addresses for their accounts in the VCS and the issue tracking system. To address this issue, we applied a matching algorithm that merges data from the issue tracking system with data from the VCS. The matching algorithm investigates the local part and the domain part of each email address separately, and merges the two data points if the local parts accord with each other. For example, we noticed that an account *username@gmail.com* is used in the issue tracking system, but not in the VCS, whereas the account *username@apache.org* is used in the VCS, but not in the issue tracking system. In this case, the matching algorithm merges the data extracted from the VCS with the data from the issue tracking system, since the local parts of the email addresses match. To avoid as many false merges as possible, the matching algorithms only merges two accounts, if one account is exclusively used for the VCS, while the other account is exclusively used for the issue tracking system.

In a second step, we acquired a list of all PMC members' repository accounts from the Apache website[5], and subsequently filtered out all the activity data that we could not associate with any PMC member. Furthermore, we also filtered out all PMC members that could not be associated with a total relative project activity of greater than zero percent, to remove PMC members that can not be classified due to missing data. The resulting absolute threshold for this activity filtering depends on the overall number of activities in the according software project. After these two preprocessing steps, our activity dataset contained 542 entries in total that could be associated to 130 different PMC members.

Table 4.3.1 provides an overview about the projects we used in the study and the number of active PMC members associated with each project, as well as the number of events related to assignee, comment, commit, merge, priority and status changes.

To generate a set of profiles, we applied k-means clustering to our data, since it is efficient and computationally cheap in handling large datasets. To

[5] `http://people.apache.org/committers-by-project.html`

Project	# Assignee	# Comment	# Commit	# Merge	# Priority	# Status	# active PMCs
Accumulo	890	10681	1804	1292	154	2050	14
ActiveMQ	234	2784	1062	8	69	587	11
Ambari	273	9355	5008	28	55	7049	20
Camel	537	3291	3929	103	66	913	13
CXF	178	2612	3762	346	17	1443	13
Drill	909	3326	805	3	106	1716	11
Felix	231	2189	1000	0	7	1082	14
HBase	1651	69737	4214	3	441	11422	25
Hive	1126	32052	2162	95	126	9257	12
Jackr.-Oak	525	6063	3874	3	101	2408	18
Jena	99	1356	995	1	34	436	5
Karaf	462	2855	2028	46	33	871	7
Log4j2	140	3178	1918	18	33	579	7
Mahout	191	5072	366	5	22	1258	6
PDFBox	240	6074	1346	0	140	1193	8
Sling	398	3636	4417	2	20	1395	12
Spark	890	6240	4643	875	292	1949	17
Stanbol	50	569	542	0	7	221	6
Tika	48	2066	345	3	4	255	13
TomEE	39	604	1460	13	5	454	2
Total	*9111*	*173740*	*45680*	*2844*	*1732*	*46538*	*234*

Table 4.1: Overview of the Apache projects used for our analysis with the active PMC members in each project, as well as the number of assignees, comments, commits, merges, issue priority, and status change events.

perform the clustering, we used Weka [Hall et al., 2009], a machine learning framework written in Java. For the clustering, we used the default settings of Weka for k-means, but we did not consider all the seven attributes we retrieved from the repositories. As the changes for the *fixed in version* attribute strongly correlated with the *status* ($\rho = 0.71$) and the *priority* attribute ($\rho = 0.72$), we removed this attribute from all further analyses. Using the k-means clustering

algorithm, we constructed four clusters, as a closer analysis of different number of clusters has shown that this is the number of clusters that is neither too low so that we end up with heterogenous profiles, nor too high so that we end up with many different profiles with only minimal differences. However, it might be possible that analyzing other projects than the ones we selected for the study might lead to different clusters. Table 4.3.1 provides an overview about the characteristics of the four clusters we mined with Weka, as well as the characteristics of the entire data set. For each attribute and cluster combination, the table displays the centroid, indicating that there are big differences between some of the clusters. The table also shows the number of PMC members that are part of each cluster.

Attribute	**Full data**	**C1**	**C2**	**C3**	**C4**
Commit	10.98	70.00	28.89	19.42	6.84
Merge	8.63	88.33	68.56	2.08	2.03
Status	9.25	64.67	9.56	34.92	4.75
Comment	6.97	42.67	3.67	22.08	4.53
Assignee	8.38	60.33	5.33	25.33	5.25
Priority	6.80	69.00	3.22	30.92	2.61
# Instances	*130 (100%)*	*3 (2%)*	*9 (7%)*	*12 (9%)*	*106 (82%)*

Table 4.2: Centroids of the four activity profile clusters as well as the whole dataset, and the number of instances that were classified into each cluster.

The *C4* cluster is by far the biggest cluster, followed by *C3*, *C2*, and *C1*. The differences in the cluster size are caused by the characteristics of a certain profile within a software project. For example, the number of stakeholders that integrate source code changes into the main code base is restricted by the number of contributed patches. These four clusters provide the ground truth for the evaluation in Section 4.4 and the basis for the definition of the rule-based profile model described in Section 4.3.2.

4.3.2 Phase 3a - Rule Inferring

A main goal of our research is to provide a rule-based profile description model to enable a project-independent profile analysis. Hence, we introduce a model called *SQA-Profiles*, which is used for the description of the set of profiles. The SQA-Profile model uses a nominal scale to formulate rule-based profiles about the activities of a stakeholder within a software development tool (e.g., VCS, issue tracking platform). We decided to use a nominal scale instead of relative values to foster the readability of the rules for all stakeholders of a software project. From our perspective, the understanding of a rule is especially important in case of an automatic change in the profile association, because a stakeholder might want to know why she was associated to another profile.

We determined the nominal scale based on the mined activity data. In a first step, we normalized the absolute numbers of activities of each single attribute within the projects to relative values from 0 to 1. Secondly, we plotted the relative attribute values of all projects and based on an initial visual analysis, we clustered the relative values with k-means into three clusters. To ensure to get the best fitting classification, we additionally run the k-means clustering for two and four clusters. The results showed that the initial number of three clusters is the most appropriate one for a scale across all attributes. We decided to label the resulting clusters with commonly used names: *High*, *Medium*, and *Low*.

In addition to the nominal scale, a set of functions and logical operations is used for the rule definition. The model supports the basic logical operations *and* and *or*. We use following definitions and functions in the model:

- ***H(attribute), M(attribute), L(attribute), N(attribute)***: Functions to prove if a passed attribute has a nominal value of (H)igh, (M)edium, (L)ow, or (N)o value

- ***A***: The set of all attribute names (commit, merge, status, comment, assignee, priority)

- ***SH***: The set of all stakeholders

Table 4.3.2 provides an overview about the converted nominal values for each of the four clusters found in Phase 2 (see Table 4.3.1). These values form the basis for our set of SQA-Profiles.

Attribute	C1	C2	C3	C4
Commit	H	L/M	M	L/M
Merge	H	H	N/L/M/H	L/M
Status	H	L/M	H	L/M
Comment	H	L/M	N/L/M/H	L/M
Assignee	H	L/M	M	N/L/M
Priority	H	L/M	N/L/M/H	L/M

Table 4.3: Nominal values found for each of the four SQA-Profiles

Based on the introduced nominal scale, logical operations, functions and the definitions, it is possible to formulate the following exemplary SQA-Profile:

Name: Example Rule

Rule: $\{s \in HH : H(s.commit) \wedge N(s.status)\}$

$HH = \{s \in SH : |\{a \in A : H(s.a)\}| > 1\}$

This example rule describes stakeholders with the following profile: at least two activity attributes with a *High* value, one of the *High* values must be the commit activity, and no activity on the status attribute. We additionally use the according quantity operators of the defined logical operations to foster the readability of the proposed rules.

4.3.3 Phase 3b - Initial set of SQA-Profiles

Based on the characteristics found in the converted clusters (see Table 4.3.2) and the SQA-Profiles model we derived the following project-independent profile definitions.

The Bandleader profile describes a PMC member that has a high activity in each attribute. We call it *Bandleader* because a stakeholder with this profile

keeps the music playing in a project, and it is very likely that the music stops when such a stakeholder leaves the project.

We found three PMC members in three different projects with this profile. The projects are Apache Drill, Jena, and Karaf. In the Apache Drill project, the stakeholder with the Bandleader profile has ten times more commits than the stakeholder with the second most commits. The activity data of the other two projects shows a similar picture.

The SQA-Profile of the Bandleader is as follows:

Name: Bandleader

Rule: $\{s \in SH : |\{a \in A : H(s.a)\}| = (|A|)\}$

The Integrator profile describes a PMC member that has a high merging activity in the VCS, and at least one other attribute with moderate activity. We call this profile *Integrator*, because a stakeholder with this profile primarily handles the integration of source code contributions in a software project. As part of this activity, source code has to be integrated in the VCS, and a change has to be noted in the according issue (e.g., status change or comment).

We found nine PMC members in nine different projects with the Integrator profile. The projects are Apache Accumulo, ActiveMQ, Camel, CXF, HBase, Hive, Jackrabbit-Oak, Sling, and Spark. None of these projects has a stakeholder associated with the Bandleader profile.

The SQA-Profile of the Integrator is as follows:

Name: Integrator

Rule: $\{s \in HH \cap HM : H(s.merge)\}$

$HH = \{s \in SH : |\{a \in A : H(s.a)\}| > 0\}$

$HM = \{s \in SH : |\{a \in A : M(s.a)\}| > 0\}$

The Gatekeeper profile describes a PMC member that has high activity in status changes and a moderate activity in assignee changes or commits. We refer to this profile as *Gatekeeper* for a stakeholder who decides when the status of an issue gets changed. We were able to find two variations of this profile. The difference between the variations is the activity in the attributes

assignee and commit. In some projects, stakeholders of this profile mainly take care of the gatekeeping on issue level. In other projects, stakeholders of this profile have a broader focus, and make changes in source code contributions or actively contribute own source code changes as well.

We found twelve PMC members in nine different projects with the Gatekeeper profile. The projects are Apache ActiveMQ, Camel, Felix, HBase, Jackrabbit-Oak, Mahout, PDFBox, Sling, and Stanbol. None of these projects has a stakeholder associated with the Bandleader profile, but five projects have a stakeholder associated with the Integrator profile as well. This can indicate that projects with a stakeholder associated to the Gatekeeper profile also have a stakeholder associated to the Integrator profile.

The SQA-Profile of the Gatekeeper is as follows:

Name: Gatekeeper

Rule: $\{s \in AA \cup AC : H(s.status)\}$

$AA = \{s \in SH : H(s.assignee) \vee M(s.assignee)\}$

$AC = \{s \in SH : H(s.commit) \vee M(s.commit)\}$

This rule covers both variations (assignee changes, commits). The threshold for one of both variations is defined with a *High* or *Medium* activity in the according attribute.

The Onlooker profile describes a PMC member that only occasionally contributes to the VCS and the issue tracking platform of a project. The sporadic activity in VCS and issue tracking platforms make it hard to define a rule for this profile. We use the term *Onlooker* because, from the perspective of the VCS and issue tracking platform, their contribution is limited. However, it can be that the according stakeholders are more focused on the non-technical part of project management, such as community management. We found 106 PMC members almost equally distributed across all projects with the Onlooker profile.

The SQA-Profile of the Onlooker is as follows:

Name: Onlooker

Rule: $\{s \in M1 \cup (M0 \cap L1) \cup (L1 \cap NA)\}$

$L1 = \{s \in SH : |\{a \in A : L(s.a)\}| > 1\}$

$M0 = \{s \in SH : |\{a \in A : M(s.a)\}| > 0\}$

$M1 = \{s \in SH : |\{a \in A : M(s.a)\}| > 1\}$

$NA = \{s \in SH : N(s.assignee)\}$

We were not able to extract a clear activity pattern for this profile, but we found out that stakeholder with this profile have a certain level of activity in multiple attributes. Therefore, we described this profile with three variations addressing the activity level. The first variation addresses stakeholders that have at least two attributes with a *Medium* activity. The second variation addresses stakeholders that have at least one attribute with a *Medium* activity and at least two attributes with a *Low* activity. The last variation addresses stakeholders that have at least two attributes with a *Low* activity and no activity on the assignee attribute.

4.3.4 Phase 4 - SQA-Profiler

The proposed nominal scale and the SQA-Profiles enable an automatic processing of software development activity data. We implemented a framework called *SQA-Profiler* to automatically extract stakeholders with an activity history that matches one of the defined SQA-Profiles.

Figure 4.2 depicts the dataflow in the SQA-Profiler framework. The framework expects stakeholder records with absolute activity data as input (e.g., one commit, five comments, no merge). It also supports an automatic merging of incomplete data sets (e.g., in case that a stakeholder uses different email addresses in the VCS and the issue tracking platform). In a second preprocessing step, the absolute activity data gets normalized per project to compute the borders of the nominal values. Afterwards, every relative value gets transferred into the according nominal value. The resulting nominal values are used for the evaluation against the proposed set of SQA-Profiles.

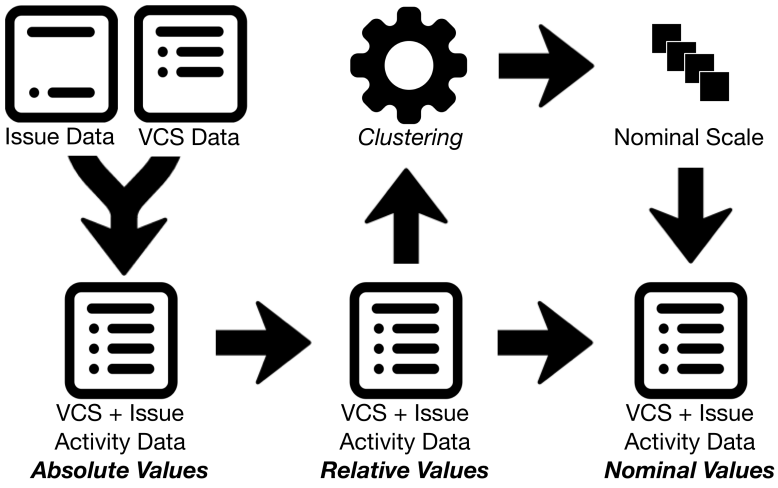

Figure 4.2: Dataflow in the SQA-Profiler

SQA-Profiler evaluates each data set against the rules of the profiles specified in Section 4.3.3. The evaluation goes from the most specific profile (the Bandleader) to more generic ones (the Onlooker). The first matching rule stops the evaluation process, and classifies the data set with the according profile. A data set is marked as unclassified in case that no rule of no profile matches it. The rules and the according evaluation are hard-coded at the current stage for simplicity reasons. In future versions, SQA-Profiler will offer a domain specific language for rule specification.

The output generated by SQA-Profiler is a list of stakeholders with their SQA-Profile based on the activity data. The SQA-Profiler is available for download on our project website.[6]

4.4 Evaluation

A central claim of our approach is that rule-based activity profiles, such as SQA-Profiles, can be used to establish project-independent profile definitions. State-of-the-art approaches, such as machine learning, provide a powerful

[6]http://www.ifi.uzh.ch/seal/people/brandtner/projects/sqa-profiles.html

tooling to cluster data precisely, but it is hard to define project-independent profile definitions. In this evaluation, we compare the results of the automatic profile association provided by SQA-Profiler with the semi-automatic profile association based on clusters provided by Weka. Figure 4.3 depicts a simplified overview of our evaluation method.

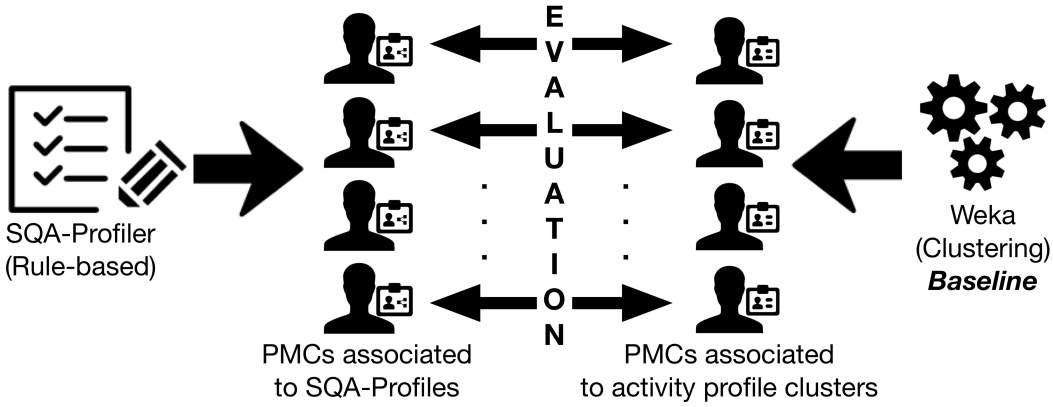

Figure 4.3: Evaluation - Overview

The input data for both approaches is a list of PMC members and their associated activity data. In case of the SQA-Profiler approach, the initial input data contains data sets of non-PMC members as well. The filtering of PMC members takes place after the preprocessing steps. This is necessary, because the transformation of the absolute activity attribute values to nominal values takes place in the preprocessing steps. A transformation without the activity data from non-PMC members would distort the nominal values. Figure 4.4 depicts the data-flow of the SQA-Profiler in this evaluation setting.

We ran the evaluation on the activity data of 20 Apache projects and automated the evaluation process to cope with the large amount of data. An evaluation program starts (1) a Weka instance for clustering, (2) a SQA-Profiler instance to associate profiles, and (3) compares the stakeholder-profile associations per project. The Weka instance is started with a data set that was manually preprocessed up-front. The preprocessing incorporates the merg-

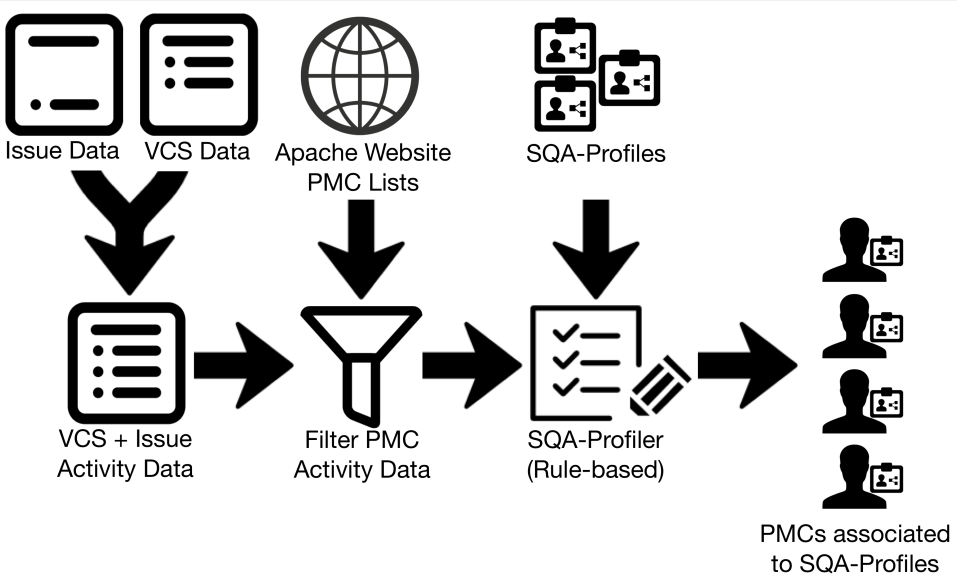

Figure 4.4: Evaluation - Data flow

ing of different identities used in the VCS and issue tracking platform. The SQA-Profiler instance uses raw activity data extracted from the VCS and issue tracking platform as input.

Table 4.4 lists the precision, recall and F-measure achieved by our automatic and rule-based approach compared to the semi-automatic baseline with machine learning. A *true-positive* (TP) is any stakeholder-profile association that is in accordance with the classification of the baseline dataset and a *false-positive* (FP) is any stakeholder-profile association that is not part of the baseline dataset.

In total, our approach classified 101 stakeholders correctly (*true-positive*), 9 stakeholders to a wrong profile (*false-positive*), and 20 stakeholders kept unclassified (*false-negatives*). These results lead to an overall precision of 0.92 and a recall of 0.78 compared to the baseline.

The Integrator profile achieved the best result with a precision of 0.90 and a recall of 1. Followed by the Bandleader and the Onlooker profile with a precision of 0.75 and a recall of 1, and a precision of 0.98 and a recall of 0.75, respectively. The Gatekeeper profile has a precision of 0.64 and a recall of 0.75,

Profile	TP	FP	Total	Precision	Recall	F-measure
Bandleader	3	1	3	0.75	1.00	0.86
Integrator	9	1	9	0.90	1.00	0.95
Gatekeeper	9	5	12	0.64	0.75	0.69
Onlooker	80	2	106	0.98	0.75	0.85
Overall	*101*	*9*	*130*	*0.92*	*0.78*	*0.84*

Table 4.4: Rule-based classification - Performance

which leads to a relatively low F-measure (0.69) compared to the other profiles. A reason for this low precision can be the broad definition of this profile caused by different gatekeeping processes of different software projects. For example, in some projects the Gatekeeper changes the status and the assignee, whereas in other projects the Gatekeeper has to additionally apply the patches. The same reason affects the Onlooker profile. Another interesting point are the *false-positive* matches in the Bandleader and the Integrator profile. These two false-positives a very likely caused by the blurring, which was introduced with the conversion from numerical to nominal values in the SQA-Profiles. Based on the nominal attribute values the profile association is correct, but based on the numerical values the matches are wrong.

Additionally, we evaluated the distribution of nominal values to ensure that they are equally distributed. This is important since the proposed nominal scale determination approach does not explicitly address this issue. The chart on the left in Figure 4.5 depicts the distribution of the nominal values extracted from the activity rating of all stakeholders. Despite the share of activities with "no rating" is larger than all other shares, the figure does not indicate any unequal distribution introduced by the nominal scale.

The second chart in Figure 4.5 depicts the number of different activity ratings per stakeholder. This chart shows that three percent of the analyzed PMC members have rated all attributes with the same value. An example profile for a stakeholder with *High* rating in all attributes is the Bandleader. The number of different activity ratings per stakeholder is an important value for

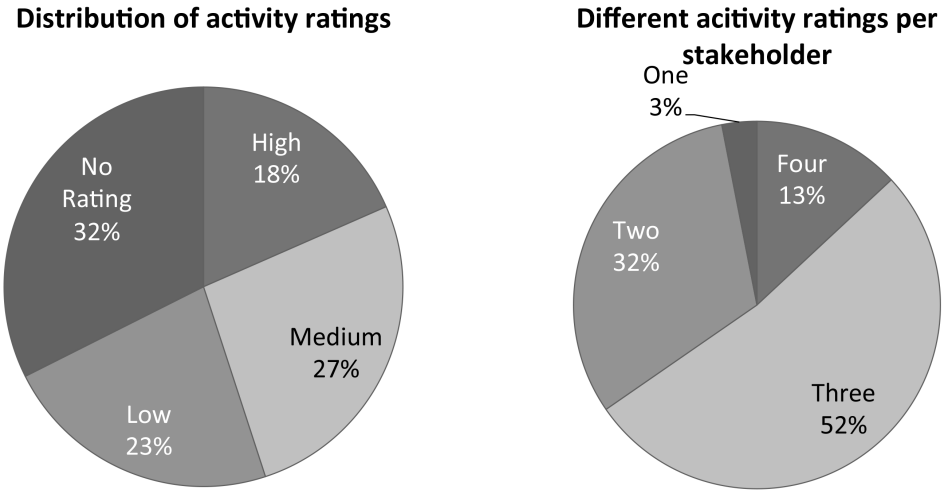

Figure 4.5: Distribution of activity ratings and number of different activity ratings per stakeholder

the interpretation of the evaluation results. For example, three of the proposed SQA-Profiles have a restriction on the minimum number of a certain attribute value (e.g., more than two attributes with a *Low* value). The result show that 65% of the stakeholders have more than two different activity ratings. This can be an indicator that the number of *false-positives* can be reduced by the use of more attribute restrictions in the according SQA-Profile definition.

We additionally evaluated the distribution of the established SQA-Profiles between PMC members and stakeholders that are not part of a PMC. Table 4.4 shows the projects, in which a certain profile was found and whether or not the stakeholder is a PMC member.

The results showed that stakeholders with the Bandleader profile are always members of the PMC. In case of the Integrator profile, the majority of the found stakeholders are PMC members as well. However, in four projects the Integrator profile is associated with a stakeholder that is not a PMC member. In all of these four projects, no PMC member was associated with the Integrator profile. With exception of the Apache Accumulo project, each project has exactly one stakeholder with this profile. The Gatekeeper profile is associ-

Profile	PMC Member	No PMC Member
Bandleader	Drill, Jena, Karaf, Spark	-
Integrator	Accumulo (2), ActiveMQ, Camel, CXF, HBase, Hive, Jackrabbit-Oak, Sling, Spark	Ambari, Log4j2, Mahout, TomEE
Gatekeeper	ActiveMQ, Camel, Felix (3), HBase, Hive, Jackrabbit-Oak, Mahout, PDFBox, Sling, Stanbol, Tika (2)	Accumulo (2), Ambari (2), Camel, CXF, Felix (2), Log4j2 (2), Spark, Tika, TomEE (2)
Onlooker	all projects	all projects

Table 4.5: SQA-Profiles per Apache project

ated with stakeholders with and without PMC membership. Different to the Bandleader and the Integrator profile, this profile is assigned to multiple stakeholders independent of their PMC membership. For example, in the Apache Felix project the Gatekeeper profile is associated with three PMC members and with two stakeholders without PMC membership. The Onlooker profile was found in all of the analyzed projects independent of a stakeholder's PMC membership. The found stakeholders are almost equally distributed across all projects.

4.5 Discussion

Overall, we found evidence that activity data mined from the VCS and the issue tracking platform can reflect the tasks of stakeholders within a certain role. The evaluation results showed that the rule-based SQA-Profile approach performs almost as good as the baseline approach using machine learning. These results indicate that our automatic and rule-based approach can achieve similar results as a semi-automatic and project-dependent approach. We discuss the benefits of a rule-based approach and a number of factors that can influence the performance of SQA-Profiles.

4.5.1 Nominal Scale & Rule-based Profiles

The proposed nominal scale and rule-based profiles provide a simple yet powerful model to describe stakeholder profiles. We showed that, despite this simplicity and the project-independent definition, the SQA-Profiles approach performs almost as good as machine learning using precise values.

From our perspective, it is important to keep the rules simple and comprehensible for stakeholders such as project managers. The rules have to be transparent and easily adoptable, because even a perfect profiling approach can produce results that are not rational from the perspective of a certain software project. This is important, because missed information in software development influences the decision quality and the project budget. Especially, a changing focus and changing activities of a stakeholder during her work on a software project can lead to non-optimal results. For example, the amount of patches increases shortly before a feature freeze deadline. In such a case, an additional PMC member might help out with the patch integration. The patch integration activity can influence the profile association (e.g., the associated profile changes from Gatekeeper to Integrator). In case of such a profile change, it is important that a stakeholder can follow the profile association process and adopt the profile, if necessary.

4.5.2 A Set of SQA-Profiles

The set of SQA-Profiles proposed in this work covers activity patterns of PMC members in 20 Apache projects over the last year. There are indicators that the set of rule-based profiles does not cover all profiles in any PMC of a software project.

The extracted activity dataset of some PMC members is relatively small or empty. For example, there are PMC members with only one comment or only one commit within the analyzed time range. Furthermore, for a small groups of PMC members, we were not able to find any activity in the VCS or the issue tracking platform. We can imagine two scenarios for the absence of activities. The first scenario is that the according PMC members no longer

participate in the development of the project. The second scenario is that the according PMC members are in charge of community related tasks (e.g., management of mailing lists, wikis) and, therefore, do not contribute to the VCS or issue tracking platform. We decided to not cover such scenarios in our set of SQA-Profiles, as it is very likely that such PMC members are not interested in CI-data.

4.5.3 Project Organization

The evaluation results showed that the existence of profiles and the number of stakeholders associated with a profile within a software project is influenced by the project's organization. For example, in projects with a stakeholder associated to the Bandleader profile, no further stakeholder was found with an Integrator or Gatekeeper profile. This is interesting because the SQA-Profile model evaluates each stakeholder independently from each other. Theoretically, it would be possible that a project has a stakeholder associated to the Bandleader profile and another stakeholder associated to any other profile. Based on the analyzed projects, a stakeholder with the profile Bandleader is an indicator that a project has a relatively small *truck factor* [Williams and Kessler, 2002]. The *truck factor* indicates the number of stakeholders that have to be hit by a truck before the projects gets incapacitated.

The results further showed that the existence of a stakeholder with the Integrator profile indicates when a software project has a dedicated source code integration process. However, the absence of a stakeholder with this profile does not indicate the absence of a dedicated integration process. In some cases, contributors hand-in patches as attachment in the issue tracker. The integration of such patches is typically done by a PMC member and from the perspective of the VCS it is hard to differentiate them from normal source code commits.

4.5.4 Project Relationships

The analysis of activity data from the VCS and issue tracking platform showed that some of the Apache projects have a strong relationship. We found relationships on technical level and on PMC member level.

We found one stakeholder that is PMC member and contributor in various Apache projects. He also appears in the PMC of five projects analyzed in this work: ActiveMQ, Camel, CXF, Felix, and Karaf. The link between the mentioned projects is the Apache ServiceMix project, which combines the mentioned projects to an integration container and where the stakeholder is PMC member as well. Despite his PMC memberships in these five projects, the stakeholder contributed only to the Apache Felix and Karaf project in the last year.

Entries in the issue tracking platform indicate that there are technical dependencies of Apache projects as well. For example, issue entry *FELIX-4436* in the Felix project is caused by an improvement in the ServiceMix project described in issue entry *SMX4-956*. Furthermore, the issue entry *KARAF-2420* in the Karaf project is related to the mentioned improvement. The initial issue entry in the Apache ServiceMix project requests an improvement in the monitoring of changes in a configuration file. This example shows that relatively simple changes in one project can affect multiple other projects.

In our work, we evaluated the performance of SQA-Profiles on project-level. We profiled activity data of stakeholders per project and independent of any activity in other projects. In order to support project-overlapping profiles and change tracking, it would be necessary to adopt the profile association algorithm and to derive additional SQA-Profiles.

4.5.5 Contributors with PMC Profiles

In our evaluation, we extracted a number of non-PMC members in Apache projects, which have an activity history matching PMC profile (see Table 4.4).

We found three projects that have associated the Integrator and Gatekeeper profile only to stakeholders that are not members of the PMC: Apache Ambari,

Log4j2, and TomEE. In case of the TomEE project, the activity of the PMC members in the VCS and the issue tracking platform is relatively low. We could only find activities of two PMC members. Most of the source code contributions originate from the contributors. The contributors also take care of the patch integration and the management of the issue tracing platform. The analysis of the Log4j2 project draws a similar picture. In case of the Ambari project, the situation is different. The PMC of the Apache Ambari project consists of 37 stakeholders, which is large compared to other Apache projects. Only ten out of 47 contributors are not PMC members. However, the contributions of the PMC members seem to be limited because the Integrator and Gatekeeper profile are associated to stakeholders that are only listed as contributor.

The existence of non-PMC members with a PMC profile can be seen as indicator that the roles assigned in a software project do not always reflect the actual activity of a stakeholder. This finding impacts our proposed approach in the definition process of SQA-Profiles, because wrongly assigned stakeholders may blur the resulting SQA-Profiles.

4.5.6 View Composition and Information Tailoring

The motivation of this work originated from the idea to automatically compose views and tailor information for CI dashboards based on activity profiles of stakeholders.

We showed that it is possible to extract activity patterns from the data of a VCS and an issue tracking platform. These patterns can be used to establish rule-based profiles for an automatic processing. The evaluation of 20 Apache projects further showed that a stakeholder profile described with our proposed rule-based model is project-independent.

However, the data-driven objective of our approach has limitations as well. A major limitation is the classification of profiles that describe stakeholders with a low activity in the used data sources (e.g., VCS, issue tracking platform). This limitation is reflected in the relatively large Onlooker profile. Due to

the small activity it is neither possible to further split up the cluster nor to extract a significant activity pattern. One possibility to overcome this limitation would be to raise the threshold of the minimum activity of a stakeholder that is required to enable the profiling. In general, we propose a generic view for all stakeholders that have no profile associated, because of a low activity. Another limitation is the assumption that activity data reflects the importance of information. For example, a change in a source file causes an error in another file, which was never touched by the according stakeholder. From the perspective of the activity data, the information about the changed source file is more important to the stakeholder than information about the other file. One approach to overcome this would be adding structural information, such as source dependencies, to the model.

Overall, we see our approach as a milestone to enable a fully-automatic data processing for information tailoring and view composition in CI environments.

4.6 Threats to Validity

Empirical studies have limitations that have to be considered when interpreting their results. Our study is amenable to threats to the external, internal and construct validity.

External Validity. For the extraction of the rule-based SQA-profiles, we relied on activity data gathered by mining source code repositories and issue tracking platforms of 20 different Apache projects. We limited the activity data extraction to a period of one year. These decisions might limit the generalizability of our results, and further studies might need to be conducted to verify that our results can also be applied to other projects. However, to mitigate this risk, we have chosen the projects used in our analysis in a way to get a broad sample of various projects with different characteristics.

Internal Validity. For the evaluation of the rule-based activity profiles, we first used Weka and applied clustering to the activity data to get four clusters that we then used as the ground truth. Thus, the clusters might only be an approximation of the real activity profile of each stakeholder, which can affect

the results of our evaluation. We mitigated this risk by verifying that the clusters are sound, that is, the data in the clusters are similar to each other while dissimilar to data of other clusters.

As another threat to the internal validity, we did not differentiate between the various projects that we used in our study and applied the same approach to all the projects. However, as the results have indicated, there are for example projects that adhere to the Apache guidelines that state how a program committee should work, while others do not. By taking these differences into account, we might be able to improve our results even more.

Construct Validity. The major goal of our approach is to establish a rule-based approach to automatically extract SQA-Profiles. This rule-based approach relies on commit and issue management activities of stakeholders involved in a software project. A threat to the validity of the study is that there might be other factors than the commit and issue management activities that have an influence on a stakeholder's focus within a certain role, which are currently not captured by our approach. Further studies need to be conducted to examine the influence of these yet unknown factors. Another potential threat of our proposed approach is the partial re-use of the activity data for the profile extraction and for the evaluation. We tried to mitigate this threat by using only the activity data of PMC members for the profile extraction and the activity data of all members of an Apache project for the evaluation.

4.7 Related Work

The proposed approach of extracting stakeholder profiles based on activity data can be seen as intersection of multiple research areas. In the following, we discuss the most relevant related work from the following areas: socio-technical networks, bug prediction, and developer context.

Socio-technical networks: Bird et al. [Bird et al., 2008] mined communication and development data, and found that strong community structures exist in the communication patterns of open source projects. Surian et al. [Surian et al., 2010] investigated patterns in a graph-based representation of developer

interactions. They used the found patterns to establish a recommendation for finding developers with similar properties [Surian et al., 2011]. Another approach proposed by Meneely et al. [Meneely et al., 2010] aims to enrich the data gathered from VCS with issue tracking annotations. Their results showed that some groups of contributors never appear in the VCS, but actively influence the development process. A further topic of this research area is social coding. Dabbish et al. [Dabbish et al., 2012] investigated the influence of visible feedback on the collaboration of community members. They indirectly categorized the roles of developers based on attributes, such as number of followers or commenting activity. The related research in socio-technical networks showed that different attributes (e.g., number of commits or comments) from multiple repositories (VCS, issue tracker, etc.) can be used to successfully model the interactions of developers within a software project.

Bug prediction: Antoniol et al. [Antoniol et al., 2008] proposed an approach to classify and distinguish bugs based on the information in the issue description. They used an alternating decision tree to predict the type of an issue. Guo et al. [Guo et al., 2010] investigated a characterization of bugs to predict which of them get fixed. For example, they showed that the number of reassignments negatively influences the likelihood for a bug fix. Zimmermann et al. [Zimmermann et al., 2012] categorized the bug reopen process based on quantitative bug data (e.g., state, assignee, type) and survey data where they asked about reasons for a reopening. Ostrand et al. [Ostrand et al., 2005] introduced a negative binomial regression model to predict the expected number of failures within a source code file. They used the fault and modification of previous releases for their predictions. Weyuker et al. [Weyuker et al., 2007] extended the approach of Ostrand et al. with developer information. They derived metrics addressing the number of developers, which modified a file. Pinzger et al. [Pinzger et al., 2008] investigated the fragmentation of developer contributions and the number of post-release failures. They established a contribution network and showed that centrality measures, such as number of authors and commits can predict failure-prone binaries with a high precision. The relation of this research area to our approach is the systematic analysis of

attributes from repositories to derive rules for bug prediction or in our case profiling.

Developer context: State-of-the-art integrated development environments (IDEs), such as Eclipse, provide various interface configurations for different roles (e.g., Java Developer, Web Developer). Findlater et al. [Leah Findlater, Joanna Mcgrenere, 2008] showed that a fine-grained and more task-oriented grouping of interface elements is more efficient compared to a single user interface composition per role. Cheng et al. [Cheng et al., 2003] investigated the collaboration data of the Rational Team Concert platform. The collaboration data can be used to support the composition of personal user interfaces in IDEs. Another aspect besides the role is the task context for recommendation systems in IDEs. Kersten and Murphy [Kersten and Murphy, 2005] proposed Mylar (initial name of Mylyn) to track task contexts in the IDE. The proposed interest model of Mylar can help developers to stay focused on a task by highlighting important artifacts. Anvik and Murphy [Anvik and Murphy, 2007,J. Anvik and G.C. Murphy, 2011] investigated the implementation expertise of developers based on the data of the VCS and the issue tracker. They came up with an automatic recommender system to support bug-triaging. Fritz et al. [Fritz et al., 2010] introduced a degree-of-knowledge model to estimate the knowledge of a stakeholder about a certain source code artifact. They found that the code a developer authors and the code with which the developer interacts are not the same. Ying and Robillard [Robillard et al., 2014] proposed techniques to store and process developer profiles for recommendation purposes. They reviewed existing recommendation approaches from movie databases and investigated potential applications in software engineering. The relation of this research area is the aim to describe the context of stakeholders within a software project.

Our approach differs from the mentioned related work, as we put the focus on the individual activities of stakeholders and not on the interactions between stakeholders.

4.8 Conclusion

CI environments have become an important information source in software development. In this paper, we introduced rule-based and project-independent *SQA-Profiles* as an instrument to support information propagation in software projects.

We analyzed the activity data of project management committee (PMC) members from 20 Apache projects and derived four SQA-Profiles: Bandleader, Integrator, Gatekeeper, and Onlooker. We implemented *SQA-Profiler* as a prototypical framework to support the automatic identification of stakeholders and SQA-Profiles based on VCS and issue tracking data. The analysis showed that reoccurring activity patterns associated with a certain task (e.g., patch integration) can be found across different software projects. However, the occurrence of these patterns is not always in accordance with the assigned role of the stakeholders.

In the evaluation, we compared the performance of our automatic approach against a semi-automatic analysis with machine learning. The results showed that our rule-based and project-independent SQA-Profiles can be used to automatically extract the profiles of PMC members with a precision of 0.92 and a recall of 0.78 compared to the dataset extracted by a project-dependent and semi-automatic approach based on machine learning.

The SQA-Profiles approach can be seen as a potential data source for future algorithms that enable automatic view composition and information tailoring in CI environments. In future work, we want to bridge the gap between the proposed SQA-Profiles and our SQA-Mashup approach for an automatic composition of CI dashboards based on stakeholder activity history. To achieve this aim, we will have to translate the focus of stakeholders described by SQA-Profiles to the according data (e.g., quality metrics, build status) provided by CI environments.

5

SQA-Mashup: A Mashup Framework for Continuous Integration

Martin Brandtner, *Emanuel Giger, and Harald Gall*
Published in the Information and Software Technology Journal, Volume 65, 2015
DOI: 10.1016/j.infsof.2014.10.004

Abstract

Context: Continuous Integration (CI) has become an established best practice of modern software development. Its philosophy of regularly integrating the changes of individual developers with the master code base saves the entire development team from descending into *Integration Hell*, a term coined in the field of extreme programming. In practice, CI is supported by automated tools

to cope with this repeated integration of source code through automated builds and testing. One of the main problems, however, is that relevant information about the quality and health of a software system is both scattered across those tools and across multiple views.
Objective: This paper introduces a quality awareness framework for CI-data and its conceptional model used for the data integration and visualization. The framework called SQA-Mashup makes use of the service-based mashup paradigm and integrates information from the entire CI-toolchain into a single service.
Method: The research approach followed in our work consists out of (i) a conceptional model for data integration and visualization, (ii) a prototypical framework implementation based on tool requirements derived from literature, and (iii) a controlled user study to evaluate its usefulness.
Results: The results of the controlled user study showed that SQA-Mashup's single point of access allows users to answer questions regarding the state of a system more quickly (57%) and accurately (21.6%) than with standalone CI-tools.
Conclusions: The SQA-Mashup framework can serve as *one-stop shop* for software quality data monitoring in a software development project. It enables easy access to CI-data which otherwise is not integrated but scattered across multiple CI-tools. Our dynamic visualization approach allows for a tailoring of integrated CI-data according to information needs of different stakeholders such as developers or testers.

5.1 Introduction

A fundamental aspect of Continuous Integration (CI) according to Fowler is visibility: "*[...] you want to ensure that everyone can easily see the state of the system and the changes that have been made to it.*" [Fowler, 2014]. The integration of a modern CI-toolchain within the development process of a software project is fully automated, and its execution is triggered after every commit. Developers or testers perceive the CI process most often only in case of a build break or

test failure. In such a case, they get an automatically generated notification, for example, via email. A developer can then fix the problem using this information. This kind of exception-driven behavior helps to detect and fix problems as early as possibly during integration runs. Even in the absence of build or test failures, modern CI-tools generate a bulk of report data with each commit. This data is scattered across the entire CI-toolchain, and analyzing it, for instance, to monitor the quality of a system, is a time consuming task. This can delay the rapid feedback cycles of CI and one of its major benefits is then not utilized.

The need for an integration of the tools used in a CI-landscape is expressed in studies that address the information needs of software developers. Questions are, for example, *What has changed between two builds [and] who has changed it?* [Fritz et al., 2010]. Typically, the way to answer these kind of questions consists of two steps. First, a developer needs to know the dates of the respective builds. Second, these build dates can then be used to investigate, e.g., commit logs, file-diff data, build details, issue details etc. However, to obtain the relevant information, a developer must access several different tools and navigate through multiple views.

Additionally, the scope covered by modern CI-toolchains is not limited to build break and test failures. Quality measurements, such as code metrics, violations of coding guidelines, or potential bugs, are computed with each integration run. They provide an immediate feedback circuit for recently committed source code changes. Currently, this immediate feedback is limited to email notifications from each CI-tool due to the missing integration along a CI-toolchain. Integration approaches, such as the work of Singer et al. [Singer et al., 1997], address only tools used by developers during the creation of source code.

We derived seven state-of-the-art tool requirements from related approaches and designed and implemented a proof-of-concept mashup for data integration, and a front-end for its representation. Our framework and platform called *SQA-Mashup* is highly extensible and integrates information from various CI-tools, such as BitBucket, GitHub, Jenkins-CI, and SonarQube. The

graphical front-end of SQA-Mashup is role-based and supports dynamic views associated with different stakeholders in the context of a software project.

We use the term *stakeholder* to refer to developers and testers as these two roles are currently supported. However, we emphasize that our approach can be extended to other roles, such as software architects or project managers. The data presented in a dynamic view can be arranged either automatically or manually, based on the information needs of a stakeholder. For example, the default developer view provides code-related measurements, such as code complexity and others, while the tester view provides testing-related measurements such as test coverage etc. SQA-Mashup further supports so-called *timeline views* that use a time axis to visualize time-related information on builds, commits, or test runs. For example, the timeline view can enable the detection of increased commit activities before the daily integration run. The increased commit activity before an integration run can be an indicator that developers, for example, do not regularly check in their source changes, which can lead to error-prone merging scenarios and build breaks.

To validate our approach we defined the following research question:

RQ: How do stakeholders perform in answering questions about software quality with SQA-Mashup compared to the use of standalone CI-tools?

To answer this research question we conducted a user study with 16 participants on the JUnit project. The results show that an integrated platform has substantial advantages: information needs covered by an integration of CI-tools lead to more accurate answers in less time compared to the standalone use of CI-tools.

This work is an extension of our research presented in [Brandtner et al., 2014], which introduced the SQA-Mashup integration approach. On top of this earlier work, we newly introduce the SQA-Mashup framework and its conceptional model used for the data integration and visualization.

The remainder of this paper is structured as follows: In Section 5.2 we introduce the SQA-Mashup approach. We describe our SQA-Mashup framework and its a proof-of-concept implementation in Section 5.3. The experimental design of our user study is described in Section 5.4. In Section 5.5 we present

our main results and discuss our insights for information needs and tooling. We discuss related work in Section 5.6 and conclude with our findings in Section 5.7.

5.2 Mashing-Up Software Quality Data

The goal of our approach is to integrate the information scattered across diverse CI-tools into a single Web-based service. The information is presented according to the information needs of different stakeholders. We aim for a fast and easy way to analyze and communicate the actual state, the current health, and the most recent changes of a system. Figure 5.1 depicts the integration of data from a CI-toolchain to present it to different stakeholders. The *CI-toolchain – Tool view* in Figure 5.1 represents the information gathering process of, for example, a software tester during a quality measurement review. Starting from a quality analysis platform (*A1*), where the tester detects an increase of convention violations, the developer navigates to the build platform (*A2*) and from there to the associated source code commits (*A3*) to locate the causes of the violations. The data accessed during such an information gathering process can be modeled as a graph as depicted in the *CI-toolchain – Graph model*. This graph models each CI-tool as a tree with nodes representing the different data available in a tool. However, during a development or testing task, such as bug fixing or a quality review, only some data of each CI-tool is relevant to solve the task. The highlighted nodes (*D1-D5*) in the tree model indicate the data accessed by the software tester during his review. Our proposed mashup integrates the single pieces of data from different CI-tools into a condensed model (*SQA-Mashup – Graph model*). It is used to dynamically represent (*SQA-Mashup – Tool view*) the data according to the information needs of a stakeholder (see Section 5.2.4).

In the following, we discuss our way of data integration and presentation. We present our mashup-based data integration that is inspired by Yahoo pipes[1].

[1]http://pipes.yahoo.com/pipes/

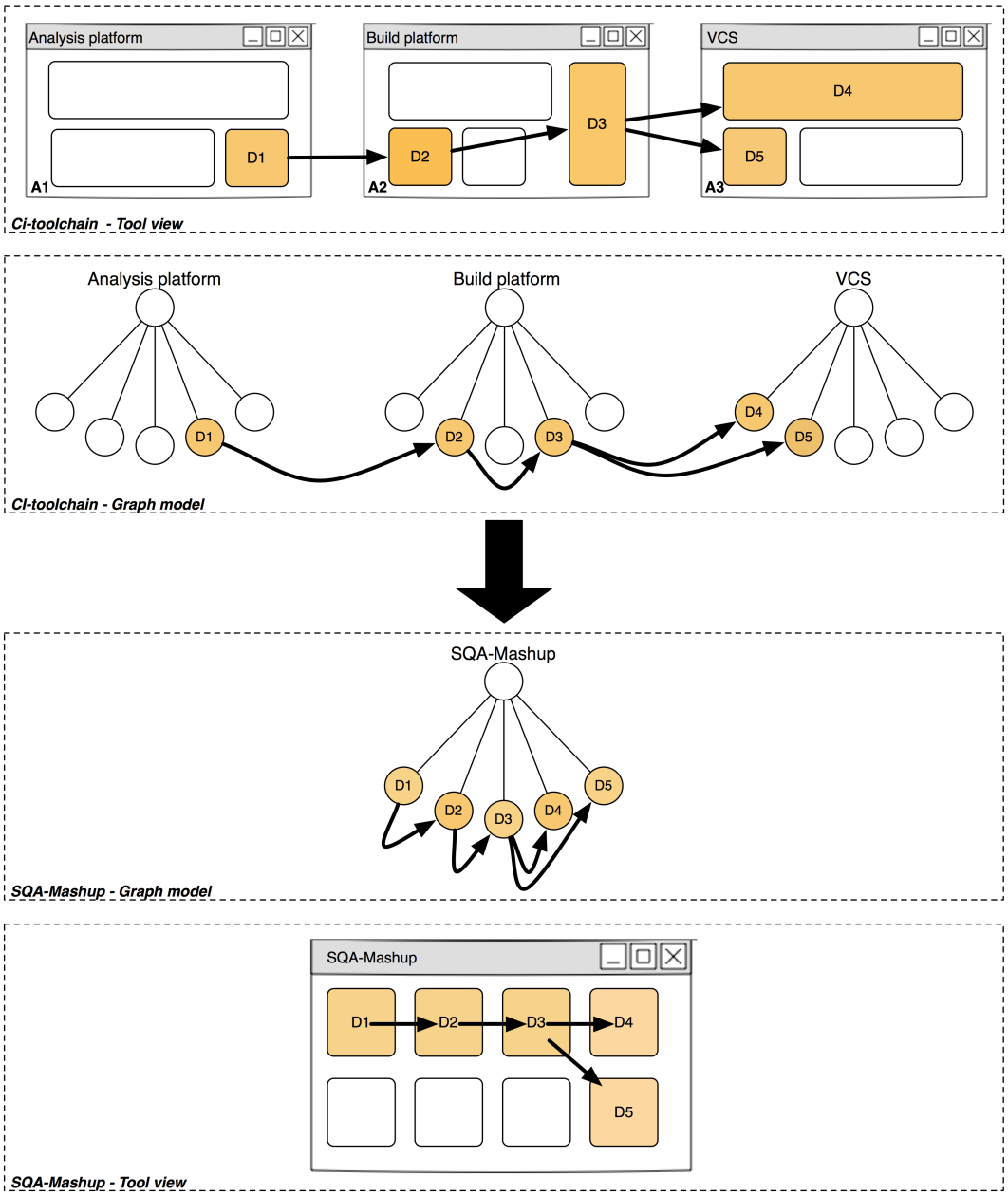

Figure 5.1: SQA-Mashup approach: Integration overview

In Section 5.2.4 and 5.2.5 we present the concepts of dynamic and timeline views.

5.2.1 Requirements

Our approach builds on a mashup-based paradigm with focus on data originating from CI-tools, which have to fulfill the following requirements:

- RESTful Web-service API (with media type JSON)
- Unique project identifier (e.g., maven artifact, etc.)

In modern CI-tools, the initial access to the data is facilitated by means of Web-Service interfaces. Our approach builds on these Web-service APIs and supports CI-tools with RESTful Web-service interface only. The integration of APIs not based on a RESTful Web-service is possible, but requires an adequate wrapper acting as Web-service.

In addition, it is necessary that each CI-tool provides a unique project identifier to enable an accurate interlinking of data across the CI-tools. An example for such a project identifier is the *artifact group* and *artifact name* when Maven is used as build automation system in a software project. Within an integration scenario it is not necessary that each CI-tool provides the same unique project identifier as a mapping between identifiers is supported by our approach.

5.2.2 CI-Tool Integration

We address the integration and interlinking of data across various CI-tools with a *mashup*, which enables a flexible selection of data from multiple data sources and to integrate them into a single consistent information space. We define so-called *integration pipes* to describe the processing steps required to interlink data from Web-services of different CI-tools. Figure 5.2 depicts an example integration pipe, which (1) collects data from Jenkins-CI and SonarQube, (2) aggregates the data, and (3) prepares the data for the visual representation in a *widget*.

The numbered arrows in Figure 5.2 illustrate the execution of an integration pipe: the execution is triggered via a request of the according Web-service

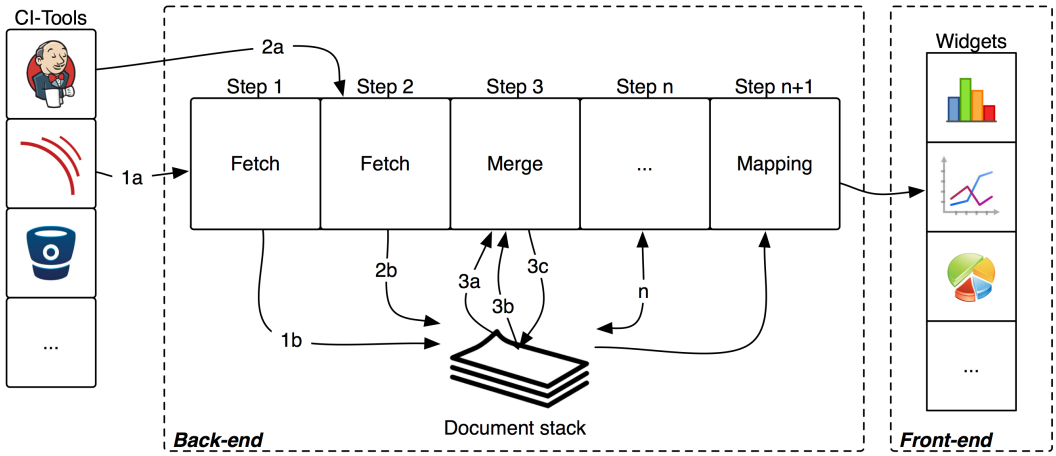

Figure 5.2: SQA-Mashup – Pipe-and-filter-based data integration

interface provided by the proposed mashup framework. We use the term *document* to address data structures, which can be used as input or output of a processing step. In this examplary integration pipe, the execution process fetches data from a SonarQube Web-service (1a) and stores it to the *document stack* (1b) of the executor thread. The document stack is used to transfer data between the independent processing steps of an integration pipe. In the second processing step, the executor fetches data from a Jenkins-CI Web-service (2a) and pushes it on top of the document stack (2b). The third processing step of this integration pipe is a so-called *merge* step. The merge pulls two documents from the document stack (3a, 3b), merges them by concatenation, and stores the resulting document to the document stack (3c). Theoretically, n further processing steps can be part of such an integration pipe, as indicated by *Step n*. The last step (*Step n+1*) pulls the remaining document from the stack, maps the retrieved data into an adequate format for the front-end, and sends the resulting document to the client, which has triggered the pipe execution.

5.2.3 Data Processing

The SQA-Mashup approach builds on the JavaScript Object Notation (JSON) as a flexible data exchange format. A standard stack approach is used for the data forwarding between the single integration steps. The combination of a flexible data exchange format and a generic document stack enables a clean encapsulation of the single processing steps.

The data processing steps that have to be supported by our SQA-Mashup approach are listed below. However, this is just an initial set of processing steps, which can be extended in future work.

- *Fetch:* The Fetch queries data from Web-services and stores it on the document stack. The expected parameters for this processing step are a URL to the Web-service and a name of an appropriate Web-service endpoint handler. An endpoint handler takes care of connection management tasks, such as authentication during Web-service queries.

- *Merge:* The Merge pulls two or more documents from the document stack, merges them, and pushes the resulting document back to the stack. The expected input parameters are two or more value-pairs. Each value-pair describes the mapping of an attribute from a source document to the resulting document with an xPath expression.

- *Mapping:* The Mapping integrates the aggregated data from different CI-tools and maps it in an adequate format for the visual representation in a *widget* (see Section 5.2.4).

An processing step that is currently not supported explicitly by SQA-Mashup is the *Transformation* of documents. For example, the use of the data fetched from SonarQube (Figure 5.2 - 1a) as an input to query data from Jenkins-CI (Figure 5.2 - 2a). In such a case, it is necessary to transform the response of SonarQube to match the input format of the Jenkins-CI query interface. At the current stage, the transformation step is part of the Fetch and the Mapping step. The Mapping as-well as the Fetch step can use documents from the

document stack as input data. The data transformation can be parameterized within the according step with xPath expressions.

5.2.4 Dynamic Views

The visual representation of our SQA-Mashup approach builds on a concept called *dynamic views*. Dynamic views allow for an integration of data from different CI-tools into screens addressing the information needs of different roles, such as developers and testers. The concept of a dynamic view is built on a graph-based dependency model for CI-data and information needs. For example, Figure 5.3 depicts Web-browser windows with CI-tools and highlights relevant data for satisfying two independent information needs of two developers (D1, D2). In addition to the information about the accessed

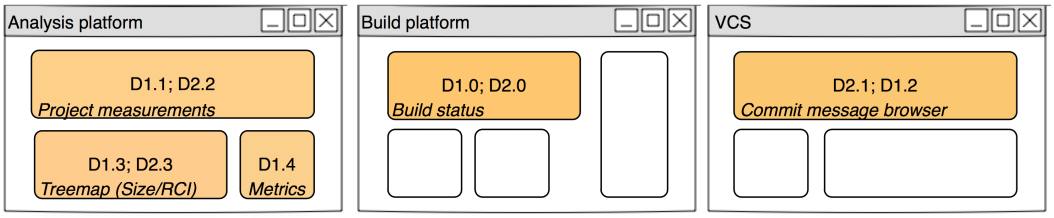

Figure 5.3: CI-toolchain – Data accessed by developer D1 and D2

data, the sequence of the data accessing flow is given by the sub-category number (e.g., Dx.1, Dx.2, etc.) as well. Based in this information we can generate a graph to model dependencies between different data from multiple CI-tools. Figure 5.4 shows such a graph, which represents the data access flow of the scenario introduced in Figure 5.3. The nodes in the graph are already arranged based on the number of dependencies for both data access flows (D1.x and D2.x). We used the valency of the nodes to express the importance of data in the information gathering process. A node with a high valency tends to be closer to the starting node compared to nodes with a small valency. In this example, we want to map the values of the graph into a view of $2\ x\ m$

widgets. A widget in our context is a graphical representation of the output of an integration pipe and can be a chart, a list or any other visualization. Based on the *2 x m* widget matrix, we restricted the space to visualize the nodes to two lanes. The node with zero inbound edges *(D1.0/D2.0)* in Figure 5.4 was chosen as starting point for our example. It represents the build status on

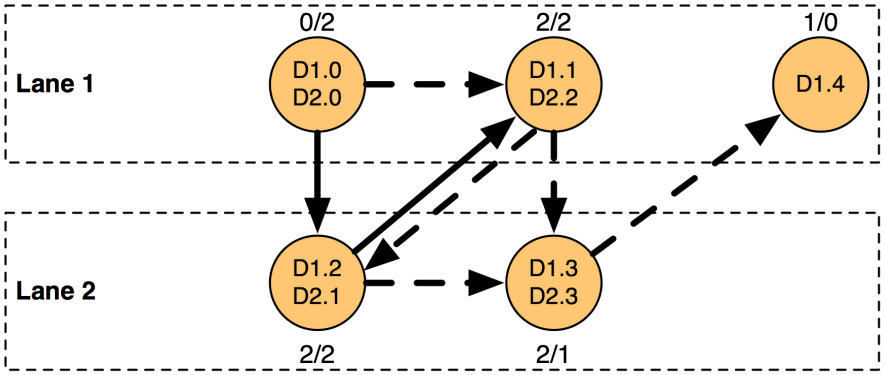

Figure 5.4: Graph – Data nodes accessed by developer D1 (- - -) and D2 (—) inclusive number of edges (in/out)

which many other CI-tools, such as quality analysis platforms, depend on for the further processing of an integration run. Based on the valency, the nodes *D1.1/D2.2* and *D1.2/D2.1* have the same importance for a developer and get accessed right after or in the second step after the build status. In our example, node *D1.1/D2.2* represents an information grid with various project measurements and node *D1.2/D2.1* represents a commit message browser. The data behind node *D1.3/D2.3* is accessed by both developers and it is the exit point for the information gathering process of developer D2. In our example, node *D1.3/D2.3* represents a tree map with information about the package size and the rule compliance. Developer D1 additionally gets information from node *D1.4*, which in our example is a detailed list of code size measurements.

5.2.5 Timeline View

The *timeline view* builds on a concept that is complementary to the dynamic view concept. CI-tools generate events (e.g., build failures), which are presented as status feeds directly in the user interface or as Rich Site Summary (RSS) feeds. Figure 5.5 depicts CI-tools with example events published over independent RSS feeds. Based on such a feed, a stakeholder can keep track

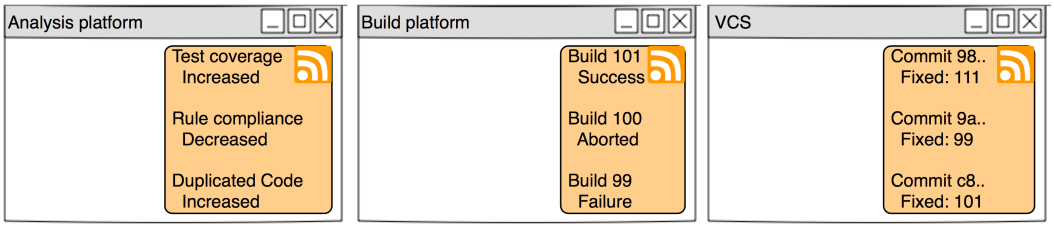

Figure 5.5: CI-toolchain – Event feeds

of what is going in a software project. The drawbacks of such feeds are the missing integration across multiple CI-tools and the missing presentation of time-related dependencies. For example, a feed of the VCS may contain the last ten commits of the current day whereas the feed of the build platform may contain the last ten builds of the last week. The feed with commits is far ahead of the feed with the build information because the frequency in which commit events occur is much higher. In such a case, a stakeholder has to imagine a time line before it is possible to interpret the information of both feeds. Figure 5.6 shows a visualization approach that provides a slightly different view on such events fired by CI-tools. The major difference to a simple feed is the representation of the timely correlation between the single events, which can help a stakeholder to better and faster understand the current situation in a software project. An integration along a time axis may foster the understanding but the view can still become overcrowded. For example, commit events occur in higher frequency than events of other CI-tools, which leads depending on the scale (see Figure 5.6) to an overlapping in the visualization.

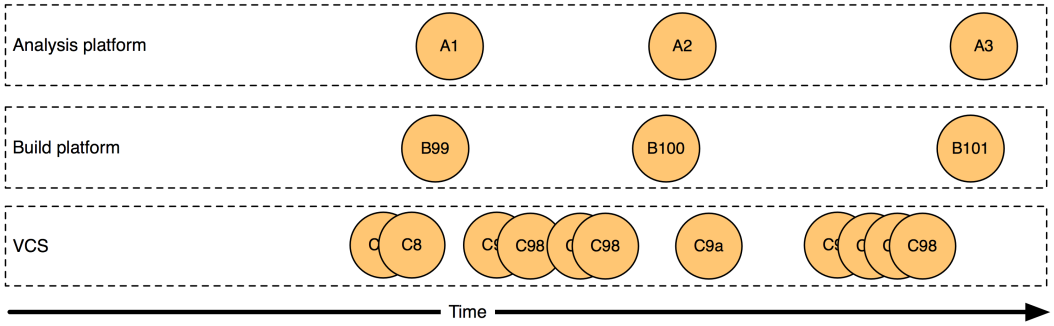

Figure 5.6: CI-toolchain – Event timeline

To address the issue of an overcrowded screen it becomes necessary to provide

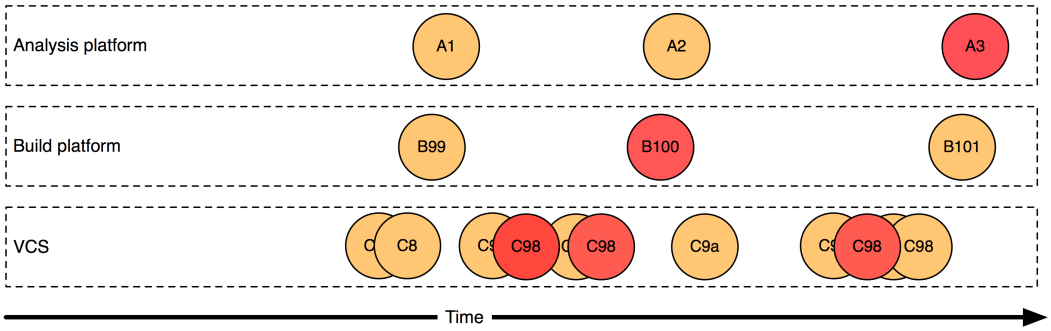

Figure 5.7: CI-toolchain – Event filter

a filter mechanism. Figure 5.7 depicts a conceptional approach to highlight certain information and filter out less important ones. The filter mechanism has to support a filtering on event attribute level to enable a selective filtering across multiple CI-tools as illustrated in Figure 5.7. For example, a stakeholder wants to locate all commits without a meaningful commit message based on a dictionary containing words such as *bug-fix*, *no comment*, *small fix*, etc. Such a commit message filter combined with a filter on failed builds can support a developer to faster locate commits without meaningful message and their influence on build failures.

5.3 SQA-Mashup Framework

In this section, we describe a proof-of-concept implementation of the SQA-Mashup framework. A service back-end integrates CI-data through integration pipes for a dynamic user interface composition (see Section 5.3.2) based on information needs. We leverage this flexibility for a dynamic visual presentation of aggregated CI-data through widgets (see Section 5.3.3). Every integration pipe of the SQA-Mashup framework back-end can be visualized through a *widget* in the front-end. A strict separation of concerns enables the visualization of one pipe output with different widgets, such as a list, a chart, or a tree-map. We pushed this flexibility even further and enabled the creation of *views*. A view is an individual arrangement of widgets according to the needs of a stakeholder. In the current implementation, we offer a view for developers and a view for testers. These two views differ in the kind and the granularity of the presented information. For example, the pipe with data about the unit tests is presented in the developer view as a chart of passed and failed test cases; in the tester view the pipe is presented as a list with more detailed information.

In the following, we discuss our proof-of-concept implementation of the SQA-Mashup approach. We discuss tool requirements and reflect the associated literature. We then present our proof-of-concept implementation of the SQA-Mashup back-end followed by a brief description of its flexible Web-service API. In Section 5.3.3 and 5.3.4 we present a proof-of-concept implementation of the SQA-Mashup front-end.

5.3.1 Tool Requirements

We surveyed the literature for tool requirements of software engineering tools. The work of Mockus et al. [Mockus and Herbsleb, 2002] and LaToza et al. [LaToza and Myers, 2010] was chosen as starting point for our survey. We investigated related work and looked for explicitly-stated and implicitly-stated tool requirements.

ID	Tool requirement	Kind	Cat
R1	Be able to locate quality hot-spots in source code (e.g. low test coverage, high complexity) Reference: [LaToza and Myers, 2010]	D	CA
R2	Permit dynamic arrangement of information shown in the user interface. Reference: [Fritz et al., 2010]	D	UI
R3	Provide awareness of the activities of peers (co-workers). Reference: [Aranda and Venolia, 2009]	D	CA
R4	Be able to discover immediately where changes occurred, when they were made, and who made them. Reference: [Mockus and Herbsleb, 2002]	E	CA
R5	Provide an interactive visualization of a person's role (e.g., developer, tester, manager). Reference: [Mockus and Herbsleb, 2002]	E	UI
R6	Be able to interoperate with other software engineering tools. Reference: [Singer et al., 1997]	E	AD
R7	Permit the independent development of user interfaces (clients). Reference: [Singer et al., 1997]	E	AD

Table 5.1: Tool requirements

Table 5.3.1 lists requirements for the integration platform presented in this paper. The column *kind* indicates if a tool requirement is explicitly (E) stated in literature or can be derived (D). We categorized each tool requirement based on the context in which it was stated: change awareness (CA) and tool-design-related requirements, such as user interface (UI) and architectural design (AD).

The entries in Table 5.3.1 show that more recent literature has a strong focus on change awareness, whereas traditionally, the focus has been on architectural design and user interfaces. The reason for this can be found in recent technological innovations, from which service-based architectures emerged. Nowadays, many CI-platforms offer a Web-service based interface to exchange

data with other systems. Tool requirement R6 reflects this functionality and requests for interoperability rather than a simple data exchange over predefined interfaces. Interoperability has to guarantee that exchanged data can be processed in each of the associated systems. Current CI-tools exhibit limited interoperability, because their service interfaces are a collection of flat data optimized for the associated user interface. A tight integration between service interface and user interface is contradictory to R7. The Web-service interface of Jenkins-CI is an example for a Web-service optimized for the default front-end. Independent development of a user interface is possible if the data layer is designed in a generic way or offers configurable services. Our definition of configurable services is described in Section 5.3.2.

A dynamic arrangement of UI-elements in state-of-the-art tools is possible, but often restricted to one configuration per project. Personalized views according to a stakeholder role and personal preferences are not supported by CI-tools, such as SonarQube. It is possible to address this issue with plugins, for example, to achieve a simple rearrangement of widgets for a personalized view. However, the capability to combine widgets from multiple views within one view is often restricted by the underlying architecture of a CI-tool. For example, a tool has one view on development data and a second view on testing data. The ability of a CI-tool to handle combined views is stated in R2. A similar tool requirement, but yet on another level, is stated through R5. The dynamic level added through R5 allows for interactive views based on a stakeholder's role. An interactive visualization adopts the view of a stakeholder based on latest changes in the data of a CI-toolchain and with respect to the affected stakeholder's role.

Agile development processes do iterations within the development and maintenance phase, compared to other software development processes. More iterations within the same amount of time require the ability to respond faster to changes happening during an iteration. CI-tools support agile software development, but each single tool restricts the data presentation to its own data. Within each iteration, a stakeholder has to navigate through every tool to become aware of changes since the last build. Tool requirement R4 addresses

the awareness of changes within each build. A similar direction is stated by tool requirement R3. The difference between tool requirement R3 and R4 is the point of view: R3 addresses the awareness between co-workers, whereas R4 addresses the awareness of changes within a certain period (for example, an iteration). R1 has a special focus on software quality measurements. Similar to R4, the aim of R1 is to become aware of changes.

5.3.2 Web-Service Presentation

The Web-service interface design plays an important role for satisfying tool requirement R7. The design of a Web-service interface aims for a clear separation of concerns and the avoidance of redundant values in response data. This is different to a user interface that provides redundant data to allow a stakeholder to easily discover dependencies between data from different CI-tools based on the tool requirements R1-R4.

We overcome the divergence between the Web-service interface description and the visual representation with a client-server architecture and a specific design of the Web-service interface (Figure 5.8). The Web-service design consists out of a *configuration channel* and a *data transfer channel* for the communication between the service and a client.

The data transfer channel contains Web-services for the actual transfer of data from CI-tools to a client application, such as a graphical user interface (GUI). Each Web-service in the data transfer channel can be dynamically created, modified, or deleted during runtime. The logic to manage the data transfer channel is encapsulated in Web-services, which are part of the configuration channel. All Web-services used in the configuration channel are hard-coded and cannot be modified during runtime. The Web-services of the configuration channel represent an API which is the starting point for a client application. Every client application with access to this API is able to consume, adopt, and extend the data transfer channel. Such a Web-service design enables an encapsulation of the complete integration logic and computation efforts into one place. This is especially interesting for concerns on change

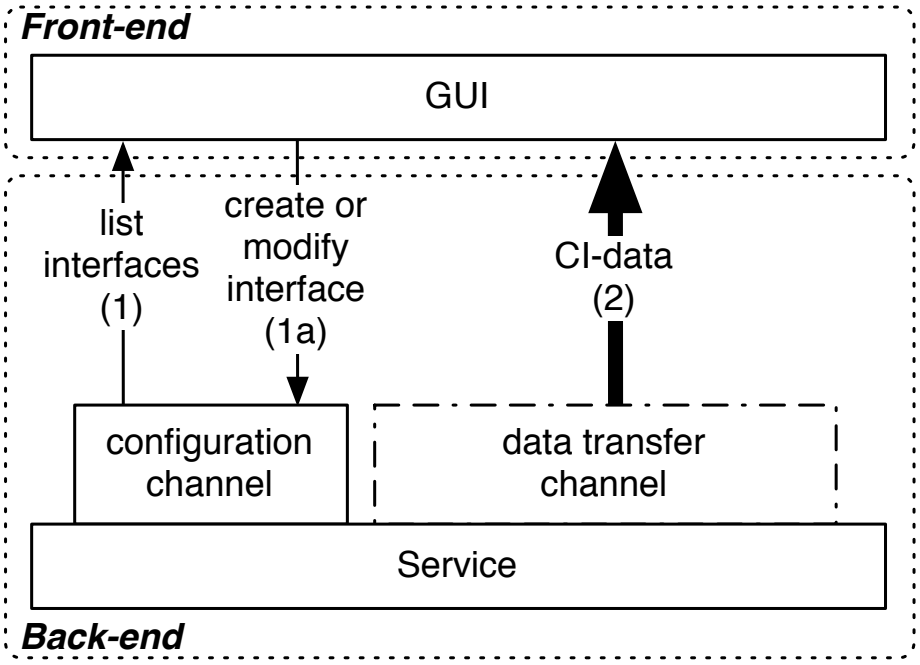

Figure 5.8: Dynamic Web-service definition through channels

awareness, such as tool requirements R2 and R4. An example is an application which visualizes the evolution of a software project based on data from CI-tools. Depending on a software project's history the amount of evolution data can be huge. For such a large amount of data our approach enables a visualization client to query meta data, which can be used to limit the actual query size. Smaller response values which accord to the needed data format save computation power on the client device. The saved computation power can be used for enhancements such as richer or faster visualization.

We use Representational State Transfer (RESTful) Web-services in combination with the JavaScript Object Notation (JSON) for the data collection, aggregation, processing, and propagation. JSON is a text-based data-exchange format with a low processing overhead and, in combination with RESTful Web-services, a de-facto standard in CI-tools.

5.3.3 Developer and Tester View

The developer and tester view of the SQA-Mashup front-end is built on the graph concept introduced in Section 5.2.4. Figure 5.9 depicts a screenshot of a role-based view with information for a software developer representing the graph nodes shown in Figure 5.4 as widgets. The figure shows different widget types and data from different information sources. For example, the *Infogrid* widget contains information about the testing (e.g., unit tests success rate) and the source code quality (e.g., rule compliance) of the SQA-Mashup project. The *Rules Compliance* widget shows a tree-map with information about the package size (rectangle size) and the rule compliance of each package (rectangle color) in the project under inspection. SQA-Mashup currently supports several well-established visualizations in the respective widgets: bar charts, line charts, pie charts, spider charts, tree maps, and lists. It is possible to extend SQA-Mashup with further and more specific widgets to visualize data from CI-tools. Furthermore, the approach of SQA-Mashup is not restricted to role-based views. It is possible to create, for example, individual views for a stakeholder as well.

The *tester view* in Figure 5.10 has a similar appearance as the developer view in Figure 5.9 but is yet different in meaning. For example, the *Test Coverage* widget looks similar to the *Rules Compliance* widget in the developer view. However, the color indicates the test coverage in the tester view instead of the rule compliance in the developer view. This is one example how dynamic views can satisfy different information needs by presenting various perspectives onto the integrated data with a similar visual appearance.

The arrangement of the widgets in a view can be done automatically via a Web-service interface or manually by drag-and-drop of single widgets into the dynamic view. In the developer and tester view the arrangement of the widgets is chosen according to the granularity and dependency of the presented information. For example, both views have the build status from Jenkins-CI and the commit data from the VCS in the first column. These data can be queried independently from the according CI-tools. The data presented in the second column can be queried independently as well but it is associated

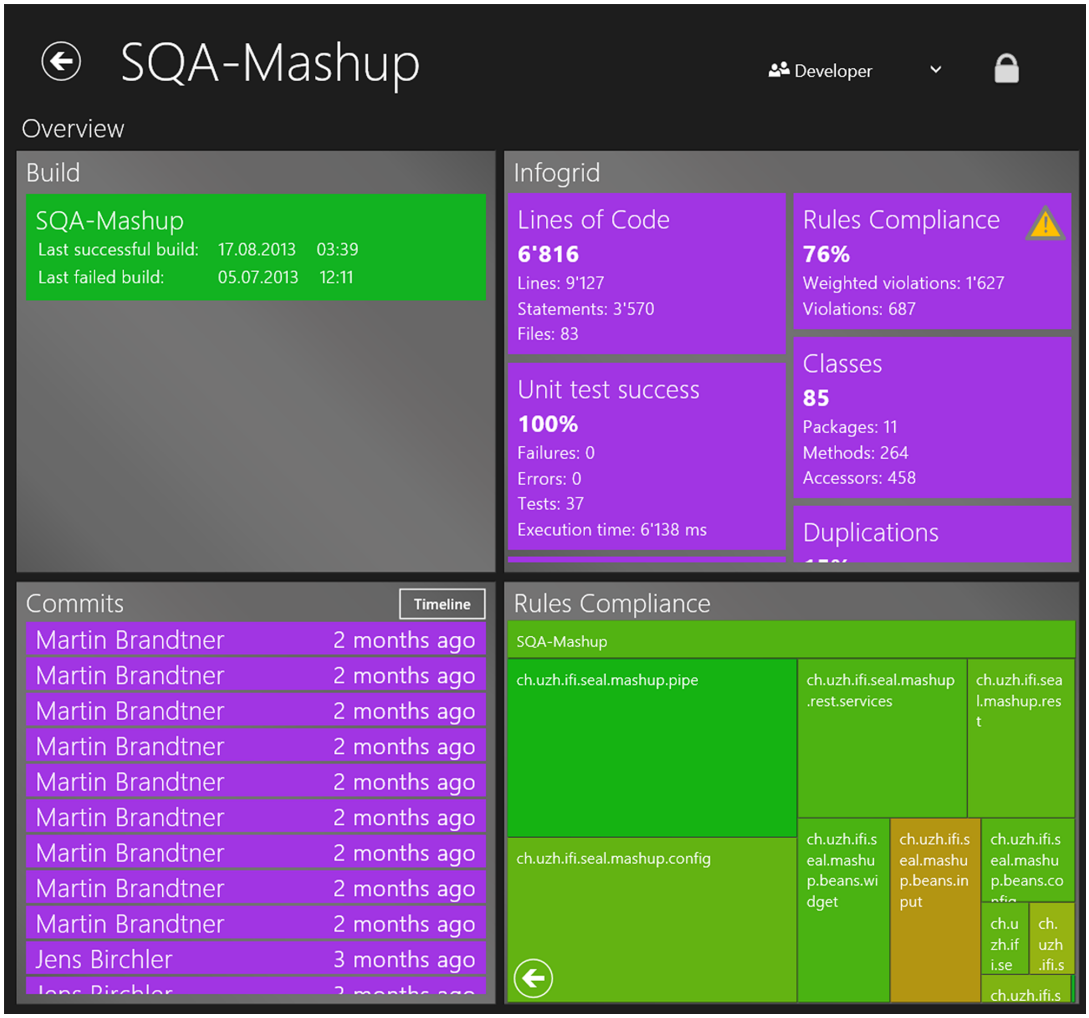

Figure 5.9: SQA-Mashup – Developer view

with the data of the first column. For example, there are no coding convention checks and unit test executions in case of a build failure because quality analysis platforms, such as SonarQube, work only with compilable source code. In case of a build failure, our approach presents the rule compliance and test coverage results of the last successful build, which is the same functionality as in state-of-the-art CI-tools. In contrast to state-of-the-art CI-tools, a stakeholder can immediately see whether or not the test coverage or rule compliance data

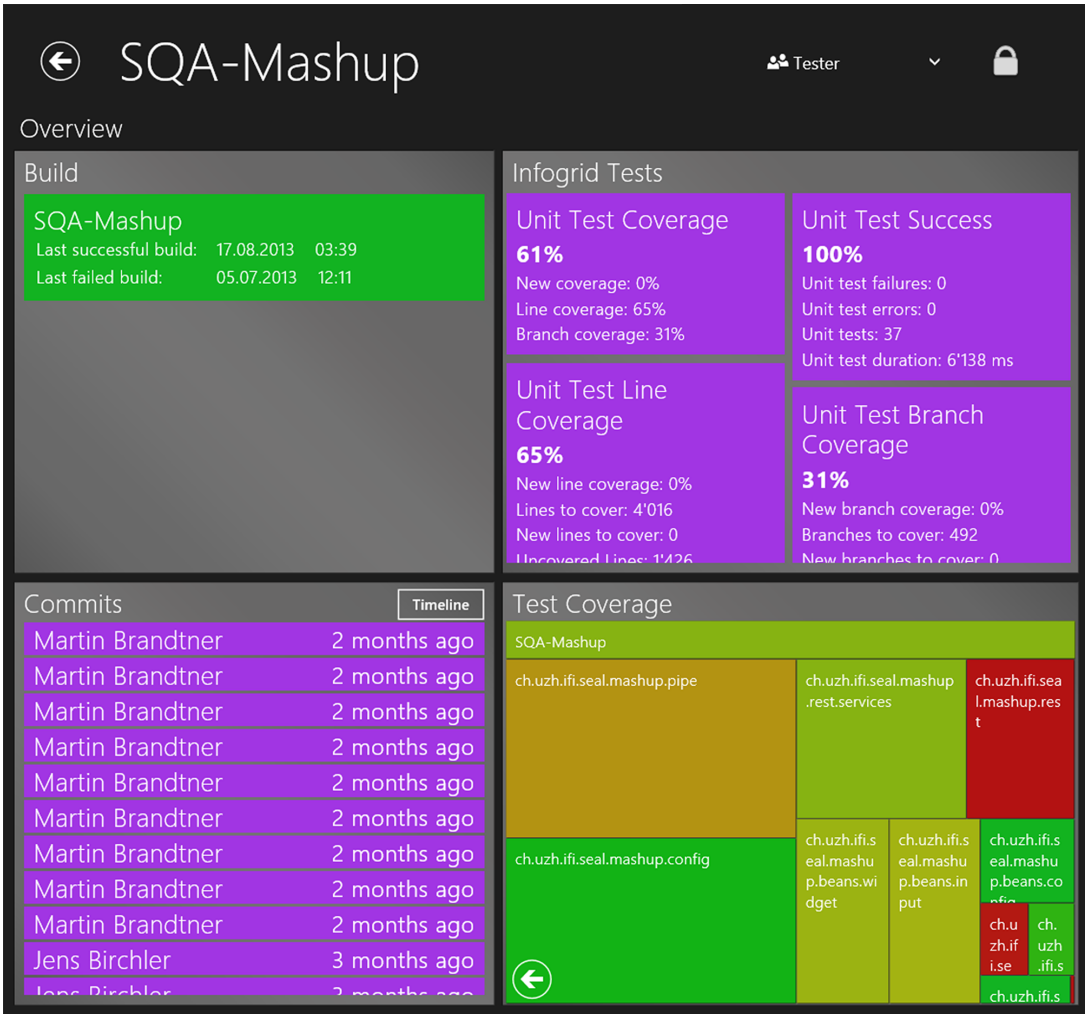

Figure 5.10: SQA-Mashup – Tester view

is up-to-date in SQA-Mashup because of the build status information, which is visualized next to the quality data.

The accuracy of quality data is essential for the work of developers, testers, and for the project management. Especially in cases when a build failure retains over a longer time period the actual test coverage may deviate from the indicated one, which was calculated during the last successful build. Therefore, it has to be assured that a stakeholder can easily verify if the data is up-to-date.

5.3.4 Timeline View

The timeline concept introduced in Section 5.2.5 is implemented in the SQA-Mashup front-end as depicted in Figure 5.11. It integrates event data from a build tool (green/red), an issue tracker (blue), and a VCS (purple). The yellow

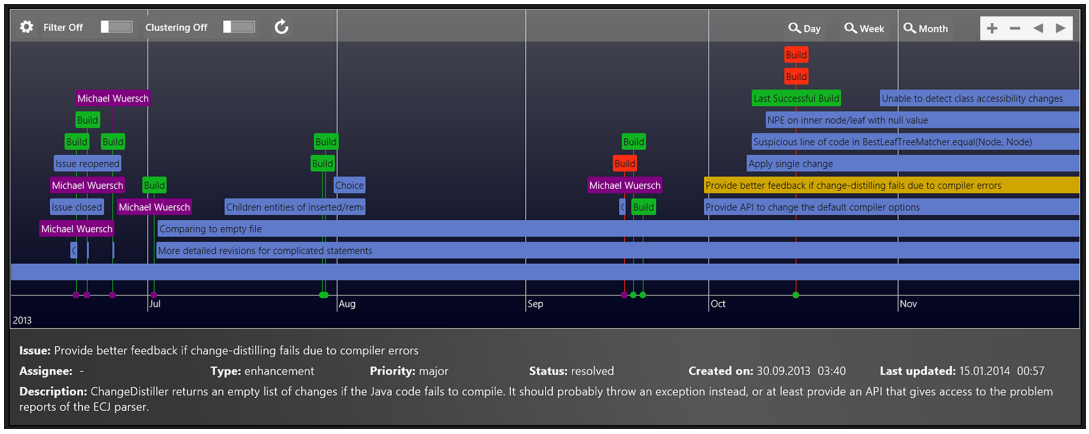

Figure 5.11: SQA-Mashup – Timeline View

bar on the right side indicates that this issue is currently selected and the detail information is shown in the bottom area of the view.

The timeline view enables a stakeholder to navigate through the history of a software project and to directly access detailed commit, issue and build data from multiple CI-tools in one screen. It is possible to filter events based on the CI-tool in order not to overwhelm a stakeholder with a large amount of data. Furthermore, a more fine-grained filtering based on the metadata of events is implemented as-well. For example, it is possible to show only commit messages which do not contain a commit message or a generic message, such as a *bug-fix*.

The data visualized in the timeline view has a different focus compared to the dynamic view. At the current stage, the dynamic views are used to visualize a snapshot of the data of the last integration, whereas the data presented in the timeline view focuses on the development history of a software project.

Of course, it is still possible to navigate to the snapshot date to see the latest events which occurred in a CI-toolchain.

The timeline view is an additional perspective to navigate through the history of a software project to better understand the temporal relation of data presented in the dynamic views. A combination of the timeline view and the dynamic views may help a stakeholder to answer information need questions, which are hard to answer with state-of-the-art CI-tools (e.g., [LaToza and Myers, 2010]).

5.3.5 Integration Pipe Configuration

In our approach, a configuration of an integration pipe is project independent and only associated to the used CI-tools (e.g. Jenkins-CI, etc.). Once an integration pipe is defined it can be used for different software projects in the same CI-toolchain. The pipe configuration of SQA-Mashup depicted in Figure 5.12 builds on the pipe-and-filter concept introduced in Section 5.2.2.

In Figure 5.12 the configuration of the *Commit Timeline GH* integration pipe is shown in the configuration editor. The *Pipe Editor* depicts a visual representation of a raw integration pipe configuration with four integration steps. The kind of widget used to visualize the data output of the according pipe (e.g., bar chart, etc.) is defined in the field *Widget Type*. A function called *Test Pipe* allows for a test run of a pipe on a software project, which is registered in SQA-Mashup to validate the integration process.

The current stage of the configuration screen is intended to be used only by professional stakeholders, which are familiar with the Web-services of the corresponding CI-tools. There is, for example, no wizard which supports the automatic joining of data from different CI-tools. A stakeholder has to manually select the fields used to join the data of two or more documents from the document stack. Nevertheless, it is possible to automate the composition of new integration pipes by utilizing the Web-service interface provided by the SQA-Mashup back-end.

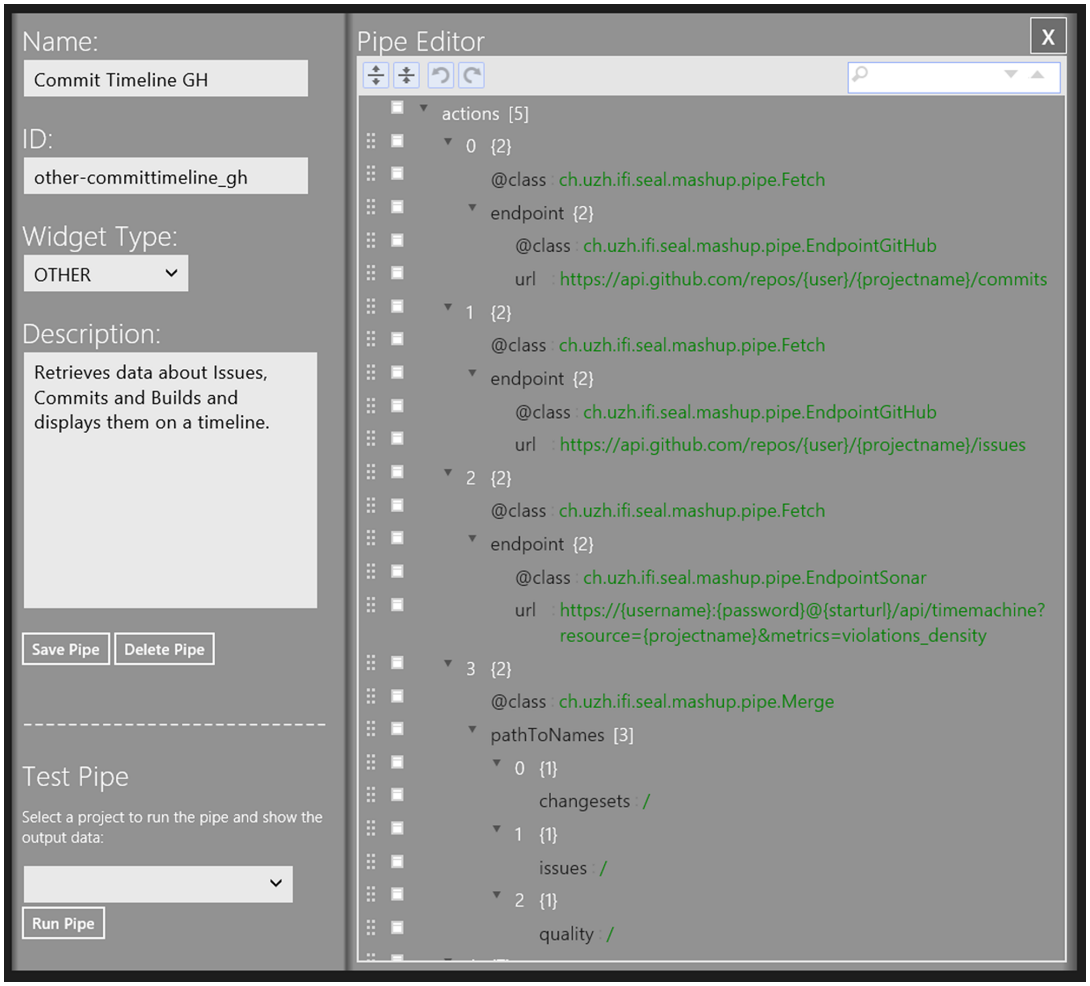

Figure 5.12: SQA-Mashup – Integration pipe configuration

SQA-Mashup is available in the Windows 8 App Store[2] and comes with a set of integration pipes to aggregate data from the CI-tools, which were used in our controlled user study (see Section 5.4). Many of these integration pipes contain not more than the two integration steps *fetch* data and *map* data into a widget. The simplicity of these integration pipes is given by the fact that the data provided by Web-services of CI-tools are close to the CI-tool's user interface. For example, the data of the overview screen in the Jenkins-CI user

[2] `http://www.ifi.uzh.ch/seal/research/tools/sqa-mashup`

interface can be fetched with one Web-service call. The largest integration pipes in SQA-Mashup are the pipes used for the data aggregation of the timeline view. In such a pipe, the event data of all CI-tools gets fetched and integrated for the visualization in the front-end.

5.4 Controlled User Study

We designed and conducted a controlled user study with 16 participants who had to solve nine software maintenance tasks. The participants were randomly assigned to either the control group or the experimental group. Each group consisted out of eight subjects and the control group confined the baseline of our study. This group had to use state-of-the-art CI-tools and a social coding platform: Jenkins-CI and SonarQube as the CI-tools and GitHub as the social coding platform. The experimental group had to use SQA-Mashup, which was implemented according to the information needs listed in Table 5.3.1. More information about the study population is provided in Section 5.4.2 and the nine tasks are listed in Table 5.4.1.

We formulated three hypotheses (see Table 5.4) and verified them through statistical tests based on the results of the user study. The hypotheses address the total values of the score, the time, and the system usability score. The total score and time are summed up over all tasks per participant.

ID	Null Hypothesis
$H1_0$	There is no difference in the total score per subjects between the experimental and the control group.
$H2_0$	There is no difference in the total time per subjects between the experimental and the control group.
$H3_0$	There is no difference in the total system usability score between the experimental and the control group.

Table 5.2: Hypotheses

5.4.1 Tasks - Nine Questions from the Literature

We exclusively selected tasks out of the software maintenance and evolution field which have been used in other studies or information need questions stated in literature. This is to ensure an objective comparison of our SQA-Mashup approach and the baseline approach with CI-tools.

We composed nine tasks which can be categorized along the domains *Program Comprehension*, *Testing*, and *Cross-Domain* questions. The tasks originate from the work of Aranda et al. [Aranda and Venolia, 2009], Fritz and Murphy [Fritz et al., 2010], LaToza and Myers [LaToza and Myers, 2010], Mockus et al. [Mockus et al., 2009], and Wettel et al. [Wettel et al., 2011].

Our aim was to preserve each task in its original structure and semantics. We applied a two-step procedure to achieve this aim. The first step was to replace all general expressions in a task with a specific entity of the software maintenance and evolution domain. For example, the question *"Is this tested?"* was refined to *"Is $Class_a$ tested?"*. In a second step we replaced placeholders such as $Class_a$ with entities from the software project used for the user study, in our case the *JUnit* project. We provide a short statement for each task to outline the reason why we selected it.

Table 5.4.1 lists the nine tasks with a short statement, categorized by their nature, and with references to the literature.

5.4.2 Study Setting

We ran the user study with 16 participants (S1-S16). Each subject had at least five years of development experience in general and minimum three years development experience with Java. Table 5.4.2 lists the degree level, the age, and the overall developer experience of all participants. Furthermore the column *Job* indicates if a participant works part-time in industry. 14 participants were advanced Master students in Computer Science. 11 out of these 14 students worked part-time in industry. The remaining two participants were researchers on PhD level. The students were recruited from courses, which were held by people other than the main authors. The control group

ID	Task
Program Comprehension Domain	
T1	**Description.** How big is the source code of $Project_a$? **Statement.** Values such as lines of code, number of classes, and number of packages allow a first approximation of the project size and structure. **Literature.** [LaToza and Myers, 2010]
T2	**Description.** The three classes with the highest method complexity in $Project_a$? **Statement.** The refactoring of complex parts of source code leads to a better maintainability of software. **Literature.** [Wettel et al., 2011]
T3	**Description.** What is the evolution of the source code in $Project_a$? **Statement.** To understand a software, it is important to know about the evolution. One way to describe the evolution of a software is to look at testing coverage and coding violations over time. **Literature.** [Fritz et al., 2010]
Testing Domain	
T4	**Description.** Is $Class_a$ tested? **Statement.** Untested parts of source code does potentially increase the risk of bugs. **Literature.** [LaToza and Myers, 2010]
T5	**Description.** Which part of the code in $Project_a$ takes most of the execution time? **Statement.** The performance of a system can be influenced by a single piece of code. The execution time of unit test can be used to find such pieces with a weak performance. **Literature.** [Mockus et al., 2009]
T6	**Description.** Which three packages have a test coverage lower than the overall test coverage in $Project_a$? **Statement.** *"Code coverage is a sensible and practical measure of test effectiveness."* **Literature.** [Aranda and Venolia, 2009]

Table 5.3: User Study - Tasks (1/2)

and the experimental group consist of eight participants each. Every subject was randomly assigned to either one of both.

ID	Task
Cross-Domain	
T7	**Description.** What are the builds between the status of $Issue_a$ has changed? **Statement.** The status changes of an issue can be used to track recurring bugs with each build. **Literature.** [Aranda and Venolia, 2009]
T8	**Description.** What have my coworkers been doing between $Date_a$ and $Date_b$? **Statement.** Quickly assessing the current status in a software project can support context switches between multiple projects. **Literature.** [Fritz et al., 2010]
T9	**Description.** What has changed between $Build_c$ and $Build_d$? Who has changed it? **Statement.** Finding changes which introduced a bug but still let the source code compile can cause substantial effort. **Literature.** [Fritz et al., 2010]

Table 5.4: User Study - Tasks (2/2)

Our SQA-Mashup approach allows for an integration of data that originate from a CI-toolchain. The tools used in a CI-chain can vary between software projects. As the baseline of our study, we decided to stay with two major open source developer communities and selected the CI-tools used in the Eclipse and Apache community. The Apache Software Foundation (ASF) provides Jenkins-CI and GitHub repositories for all projects under the ASF Development Process. SonarQube provides an instance[3] which regularly performs quality analysis of Apache projects. The Eclipse Foundation provides Hudson-CI and GitHub repositories for all projects under the Eclipse Development Process. Therefore, we decided to use Jenkins-CI, GitHub, and SonarQube as baseline CI-tools in the control group of our study. Hudson-CI and Jenkins-CI are similar in their usage as they are forks of the same codebase. The experimental

[3] `http://nemo.sonarqube.org/`

Subject	Grp	Deg. Level	Age	Dev. Exp.	Job
S1	E	PhD (CS)	32 y	8 y	N
S2	E	PhD (CS)	27 y	7 y	N
S3	E	MSc (CS)	27 y	6 y	Y
S4	E	MSc (CS)	40 y	5 y	Y
S5	C	MSc (CS)	24 y	5 y	Y
S6	C	MSc (CS)	26 y	10 y	Y
S7	C	MSc (CS)	24 y	5 y	Y
S8	E	MSc (CS)	24 y	5 y	N
S9	E	MSc (CS)	29 y	8 y	Y
S10	C	MSc (CS)	25 y	5 y	Y
S11	C	MSc (CS)	26 y	5 y	N
S12	C	MSc (CS)	26 y	5 y	N
S13	E	MSc (CS)	34 y	5 y	Y
S14	E	MSc (CS)	25 y	6 y	Y
S15	C	MSc (CS)	27 y	8 y	Y
S16	C	MSc (CS)	22 y	5 y	Y
median	C		25.5 y	5.0 y	
median	E		28.0 y	6.0 y	
SD	C		1.6 y	1.9 y	
SD	E		5.3 y	1.3 y	

Table 5.5: Study participants

group of our study had to use the SQA-Mashup front-end. The input data was gained from the same CI-tools instances used by the control group.

Each participant of our study had to solve nine tasks (described in Section 5.4.1) in the context of the JUnit project hosted on GitHub. JUnit [Louridas, 2005] is a popular unit testing framework for source code written in Java. The project is under active development since a decade. The JUnit project team uses the built-in issue tracker of GitHub to manage bugs and feature requests. At the time of our study the JUnit project on GitHub had in total 1650 revisions, about 30k lines of code, 54 contributors, and 736 issues. We decided to select the JUnit project because of the mature stage of the project. At the time of our study the commits indicated that most of the source code changes address

bugs or refactoring tasks. Only a small set of commits introduced new features. We think it is therefore a good example for a software project which is in its maintenance phase.

The maximum time to work through all tasks of our study was 45 minutes with maximum of five minutes per task. After five minutes a participant had to stop to work and to go on with the next question. The execution of one study in total lasts approximate 40-60 minutes as there was no time restriction on the introduction and the feedback section.

5.4.3 Usability

We were particularly interested in the usability of the CI-tools when the study participants solved the tasks. We decided to use the widely-known *System Usability Scale* (SUS) by Brooke [Brooke, 1996]. The SUS is a simple and easy understandable scheme which consists out of ten questions each rated on a scale from "Strongly Agree" to "Strongly Disagree." The subjects of the control group were asked to give a cumulative feedback for all CI-tools they used during the study and the experimental group rated SQA-Mashup.

As part of our user study each participant was asked to rate the *Difficulty* of each task right after solving it. We used a scale from "Very difficult" to "Very easy" in which a participant provides a personal opinion. This additional information should allow us to sharpen our understanding of the usability rating.

5.4.4 Performing the Study

We performed the user study in four sessions in the lab of our research group in Zurich. The setup for the experimental group was a computer with Windows 8 and the SQA-Mashup front-end installed on it. The setup for the control group was a standard Web-browser such as Mozilla Firefox or Google Chrome. The subjects of the experimental group started at the project page of JUnit in the SQA-Mashup front-end. Participants of the control group started with

three Web-browser windows. One window for Jenkins-CI, one for SonarQube, and one for the project page on GitHub.

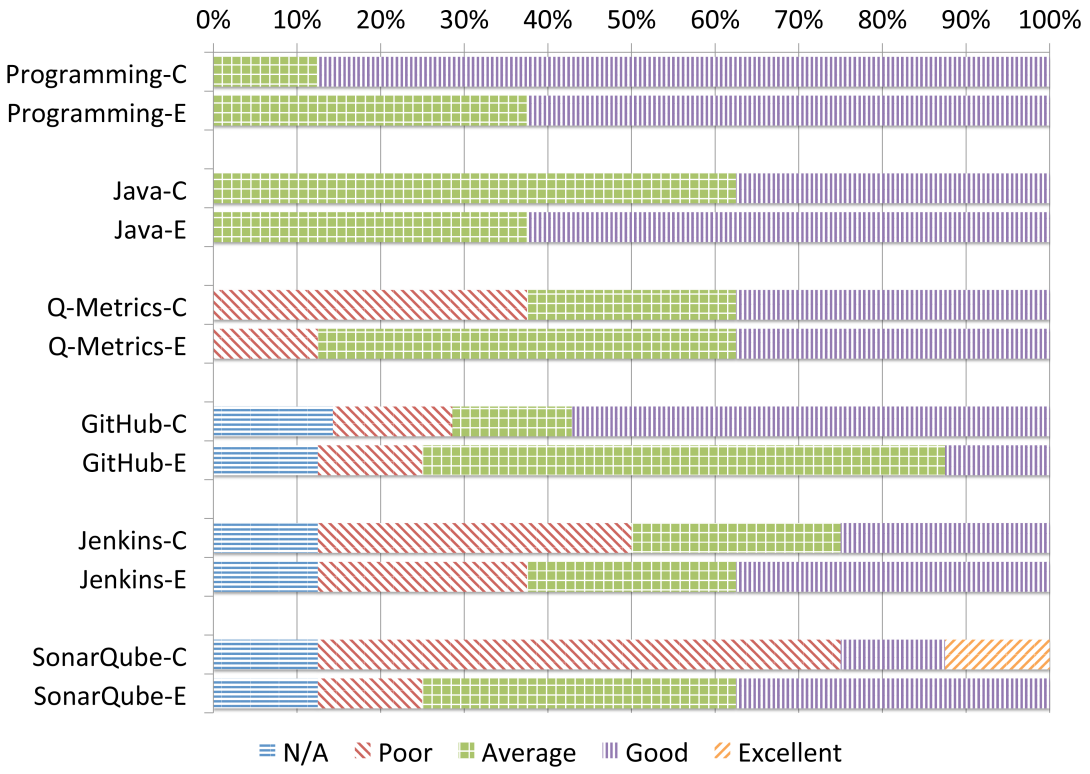

Figure 5.13: Self-assessment - (C)ontrol group and (E)xperimental group

At the beginning of every session we explained the structure of the study in addition to the written explanation on the questionnaire. The participants solved the tasks independently and the time restriction was enforced with individual stopwatches. We asked the participants of the user study to do a self-assessment of their Programming skills as-well-as their working experience with the CI-tools used in the study. Figure 5.13 depicts the self-assessment results of the control group and the experimental group. Only one participant had no working experience with any of the used platforms such as GitHub, Jenkins, and SonarQube. All others had experience with at least two out of

these three platforms. The self-assessment results of the experimental group and control group were statistically not significantly different.

5.4.5 Data Collection

The answers of the study participants were manually preprocessed for an analysis of the data with R [R Development Core Team, 2011].

The questionnaire of the user study consists of single choice questions, open questions, and rating scales for usability related questions. Every answer of the questionnaire was mapped to a numeric representation to run a statistical analysis on it. We used the following rules for the mapping:

- Rating scales: The index of the selected entry is the numerical representation, starting with zero for the first entry.
- Single choice and open questions: A grading scheme based on points with 3 points for a correct answer, 2 points for a partly correct answer (for open questions only), 1 point for a wrong answer, and 0 points for no answer.

We graded an answer of an open question as partly correct if it was correct but incomplete. For example, if the correct answer consists of two artifacts but only one was found by a participant. In case of an open question with only one correct answer the grading is the same as for single choice questions.

5.5 Empirical Results

In this section we statistically examine the results of the user study with respect to the hypotheses as listed in Table 5.4. In Section 5.5.1 we analyze $H1_0$ and $H2_0$ using the aggregate results: total score per subject, total time per subject, and the ratio of total score to total time per subject aggregated over all nine tasks. In Section 5.5.2 we analogously analyze $H1_0$ and $H2_0$ on the level of each individual task. The results of the SUS and the difficulty ratings are discussed in Section 5.5.3.

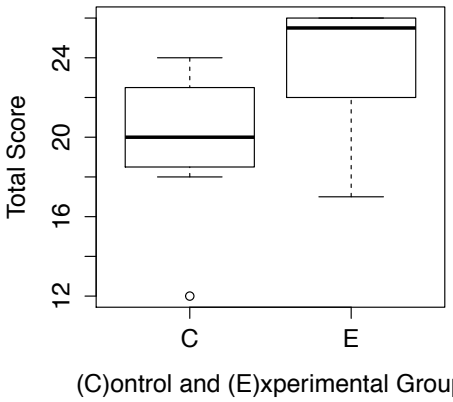

Figure 5.14: Total Score

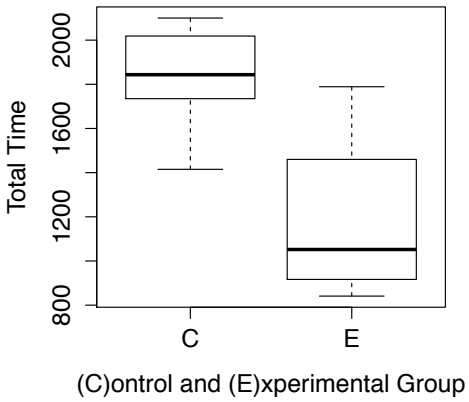

Figure 5.15: Total Time

5.5.1 Analysis & Overview of Aggregated Results

Total Score: Table 5.5.1 lists the total score per subject aggregated over all tasks for both, the experimental and control group. Each of the nine tasks was graded with 3 points if solved correctly (see Section 5.4.5), so the maximum score a participant could achieve was 27. The median of the total scores was 20.0 points in the control group and 25.5 points in the experimental group. In other words, this median difference of 5.5 points means that the experimental group outperformed the control group on average by 20.4% (=5.5/27) when referring to the total score per subject. The minimum total score per subject was 18 points (S12) and 17 points (S4), the maximum was 24 points (S10) and 26 points (S1-3,14) in the control and the experimental group, respectively. None of the participants solved all tasks correctly. One outlier (S12) with 12 points in the control group solved only four tasks - but those correctly. We could not observe any particularly obvious reasons, such as technical difficulties, tool failure, missing experience with the tools, or misunderstandings of the tasks, for this comparably low score and hence did not exclude subject S12 from our analysis.

The box-plot of the total score per subject (see Figure 5.14) of the experimental group shows that the 0.75 quantile and the median are close and almost

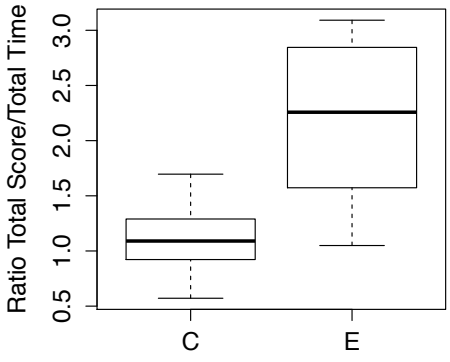

Figure 5.16: Ratio Total Score/Time

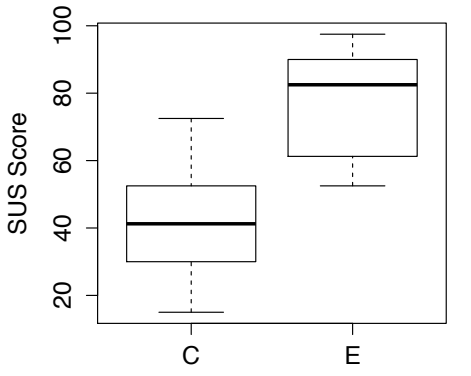

Figure 5.17: SUS Score

the same as the upper whisker of the plot. This indicates a skewed distribution. Furthermore, Q-Q plots and Shapiro-Wilk tests showed significant evidence against normality in the data. We made similar observations regarding the other quantities (see Figures 5.14-5.17) although the departures are less. Since our group sample sizes are below the commonly accepted rule of thumb (at minimum 30 to 40) we chose a non-parametric, independent-two-samples procedure, i.e., Mann-Whitney-Wilcoxon (MWW), to test all hypotheses. Generally speaking, MWW determines if the values in the experimental group are (significantly) different by comparing the mean ranks of the values in both groups. If both groups come from identical distribution they have equal chances for high and low ranks, and there is no difference in their mean ranks. Note, that the actual test itself is based on the U statistic [Mann and Whitney, 1947]. The box-plots further reveal that distributions of the values between the experimental and the control group might exhibit dissimilar shapes. In this case, MWW is interpreted as a test of stochastic equality [Mann and Whitney, 1947] rather than a test of medians. Although MWW test is a non-parametric test and can handle non-normality as well as extreme data it still makes the assumption that the distributions have somewhat similar shapes and variances under the null hypothesis. Our data indicates that these assumptions might

not be completely met. However, MWW test is fairly robust against deviations from the assumptions, and we consider this as a minor threat to validity.

Under the above considerations $H1_0$ (see Table 5.4) states that the mean ranks of total score per subject of the two groups are equal. A two-sided MWW test resulted in a p-value of 0.038 and 0.95 (non-parametric) confidence interval bounds of [1, 7]. This gives us significant evidence against $H1_0$: The total scores per subject in the experimental group (mean rank = 11) are significantly different than for the control group (mean rank = 6.0). In other words, based on the p-value and direction of the difference (as indicated by the interval bounds) we find significant evidence that the subjects of the experimental group score higher.

Total Time: Table 5.5.1 lists the total time per subject aggregated over all tasks for both, the experimental and control group. The median of the total time was 30.7 minutes in the control group and 17.5 minutes in the experimental group. This median difference of 13.2 minutes means that the experimental group outperformed the control group on average by 29.3% (=13.2/45) when referring to the total time per subject. The maximum total time per subject was 35 minutes (S11, S12), the minimum was 14 minutes (S14) and 14.2 minutes (S1) in the control and the experimental group, respectively. However, none of the participants reached the time limit of 45 minutes.

$H2_0$ (see Table 5.4) states that the mean ranks of total time per subject of the two groups are equal. A two-sided MWW test resulted in a p-value of 0.003 and (non-parametric) 0.95 confidence interval bounds of [-977, -308]. This gives us significant evidence against $H2_0$: The total time per subjects in the experimental group (mean rank = 5.12) is significantly different than for the control group (mean rank = 11.88). Based on these results we find strong evidence that the subjects of the experimental group could solve the tasks in less time.

Ratio of Total Score to Total Time: Table 5.5.1 lists the total score to total time ratio (st-ratio) per subject aggregated over all tasks for both, the experimental and control group. This value was multiplied by 100 to aid readability. The median of the st-ratio was 1.09 in the control group and 2.26 in the experi-

mental group. In other words, this median difference of 1.17 means that the experimental group outperformed the control group by about plus 0.7 points (1.17/100*60) per minute.
A two-sided MWW test resulted in a p-value of 0.005, (non-parametric) 0.95 confidence interval bounds of [0.363, 1.798], and mean ranks of 11.75 and 5.25, for the experimental and control group.

Subject	Group	Score	Time (sec)	Score/Time * 100
S1	E	26	853.0	3.05
S2	E	26	984.0	2.64
S3	E	26	1789.0	1.45
S4	E	17	1620.0	1.05
S5	C	22	1760.0	1.25
S6	C	23	1730.0	1.33
S7	C	19	1927.0	0.99
S8	E	25	980.0	2.55
S9	E	22	1300.0	1.69
S10	C	24	1415.0	1.70
S11	C	18	2097.0	0.86
S12	C	12	2100.0	0.57
S13	E	22	1120.0	1.96
S14	E	26	841.0	3.09
S15	C	20	1740.0	1.15
S16	C	20	1940.0	1.03
median	C	20.0	1843.5	1.09
median	E	25.5	1052.0	2.26
SD	C	3.73	226.88	0.36
SD	E	3.24	355.06	0.76

Table 5.6: Results per Subject aggregated over all tasks including median and standard deviation (SD)

5.5.2 Task Analysis

The aggregated results in the previous Section 5.5.1 show a significant evidence that the subjects using our SQA-Mashup approach tend to solve the tasks more correctly in less time compared to subjects using the given set of baseline tools. Correctness in our context is defined as the score achieved by a participant: A higher score means a higher correctness. The goal of this section is to further investigate where the advancements come from, and we break down the analysis of the global hypotheses H1 and H2 into individual tasks.

Due to the same reasons as on the aggregate level (non-normality and sample size below 30) the analyses on each task were performed with the MWW procedure. However, because of their post-hoc character we applied the Bonferroni-Holm [Holm, 1979] correction to the resulting p-values of the MWW tests on task level. This correction counteracts the problem of multiple hypotheses testing by controlling the family-wise error rate.

	Score (median)			Time (median)		
Task	Ctrl.	Experm.	Δ	Ctrl.	Experm.	Δ
T1	3.0	3.0	0.0	67.5	40.0	27.5
T2	3.0	3.0	0.0	112.0	67.5	44.5
T3	1.0	2.0	1.0	285.0	225.0	60.0
T4	3.0	3.0	0.0	130.0	53.5	76.5
T5	1.0	3.0	2.0	240.0	50.0	190.0
T6	3.0	3.0	0.0	125.0	79.0	46.0
T7	0.5	2.0	1.5	300.0	300.0	0.0
T8	3.0	3.0	0.0	300.0	175.0	125.0
T9	3.0	3.0	0.0	300.0	150.0	150.0

Table 5.7: Results per Task

Task T1 – Learning about project size

All participants in both groups were able to find this information and solved the task correctly. Moreover, a large adjusted p-value, such as p>0.2, clearly indicates that the observed time difference between the two groups is not significant.

Information about the project size is readily available in a particular view of the CI-toolchain. Hence not surprisingly, all participants of control group were able to write down the correct answer. We hypothesize that the single point of access for all information as provided by SQA-Mashup allows to spot the project size immediately and might offered the experimental group a head start. However, currently the results are not statistically conclusive and further work is needed on that issue, for instance, including additional participants.

Task T2 – Finding exceptionally complex source code

Every participant of both groups solved the task correctly. With respect to time a large adjusted p-value, such as p>0.2, shows that the measured time difference between both group is not significant. Again, analyzing additional subjects increases power of the test and would give more confident estimates and insights whether the observed times differences of such magnitudes in this task are due to chance.

Task T3 – Learning about quality evolution

This task was solved with a significantly higher correctness (adjusted p-value of 0.025) by the experimental group but not necessarily faster (large adjusted p-value of p>0.2).

Task T3 captures an essential activity of CI, rigorously monitoring the state and quality of a software system over time [Duvall et al., 2007]. However, to facilitate a comprehensive view of the state of the system, different quality indicators, such as test coverage, code metrics, test failures, or coding violations, must be considered. The subjects of the control group had to manually search through the individual CI-tools to locate this information. Furthermore, they

faced the difficulty that each tool uses different time periods and scales to calculate and visualize information. We generally noticed that only two subjects in the control group provided a partially correct answer, the remaining six subjects answered the task wrong. We interpret the results that a consistently integrated interface can foster project visibility and helps to encourage team awareness with respect to different software quality aspects.

Task T4 – Learning about test coverage

14 participants solved the task correctly and only one participant out of each group gave a wrong answer. With respect to time a large adjusted p-value, such as p>0.2, shows that there is no significant difference between the two groups.

Task T5 – Finding high execution times

This task was solved with a significantly higher correctness (adjusted p-value of 0.040) by the experimental group. We found a certain evidence (adjusted p-value of 0.073) that less time was needed.

While the experimental group likely benefited from the integrated view the subjects of the control group might have struggled with the fact that multiple CI-tools provide redundant information about execution times.

Task T6 – Finding test coverage below average

Only one participant was not able to solve the task correctly. Moreover, with respect to time a large adjusted p-value, such as p>0.2, shows that the measured difference is not significant.

Task T7 – Relating build information to issues

This task was solved correctly by only two subjects of the control group and four subjects of the experimental group. Generally, this task seemed time-

consuming and difficult since 75% of all participants reached the time limit of five minutes and over 50% rated it as "Very Difficult" (see Figure 5.18).

Task T8 – Learning about activities of coworkers

This task was solved significantly faster (adjusted p-value of 0.050) by the experimental group but not necessarily with higher correctness (large adjusted p-value).

The participants of the control group mentioned in their written feedback that they had difficulties in finding a specific view corresponding to the time period which we asked for in the user study. SQA-Mashup, on the other hand, offers a coherent, chronological view of developers and their activities.

Task T9 – Learning about activities between builds

This task was solved significantly faster (adjusted p-value of 0.033) by the experimental group but not necessarily with higher correctness (large adjusted p-value).

Task 9 is similar to task T8 from the perspective of builds. We expected a decrease in time to solve the task for both groups compared to task T8 because of learning effects and since builds are the central aspect of CI-tools. Only the median time of the experimental group decreased to 150 seconds, the median time of the control group remained high at 300 seconds. The scores increased in both groups.

5.5.3 Participant Ratings

We asked each participant to complete the SUS questionnaire and to rate the *Difficulty* of each task (Figure 5.18).

The result of a SUS is a score between 0 and 100 to represent the usability of a system. In our study, the score was calculated based on the original scoring scheme of Brooke [Brooke, 1996] as follows: The contribution of every item with an odd index equals the score minus one, the contribution of every

item with an even index is calculated as five minus the score. Finally, the sum of the resulting contributions is multiplied by 2.5. Figure 5.17 depicts a box-plot based on the SUS scores of the control and the experimental group. The usability median of the control group is 41.25 compared to 82.5 of the experimental group. Note, it is by chance that the usability measured in the experimental group is exactly twice of the control group's usability. The minimum whisker of the experimental group with 52.5 is slightly above the 0.75 quantile of the control group with a usability of 48.75.

$H3_0$ (see Table 5.4) states that the mean ranks of the usability scores per subject of the both groups are equal. A two-sided MWW test resulted in a p-value of 0.004 and 0.95 confidence interval bounds of [15, 55]. This gives us significant evidence against $H3_0$: The usability score per subject in the experimental group (mean rank = 11.94) is significantly different from the score per subject in the control group (mean rank = 5.06). Based these findings there is strong evidence that the subjects of the experimental group perceived the integrated SQA-Mashup front-end more ”user friendly”.

Next, we describe differences in the individual task ratings regarding *Difficulty* between the control group and experimental group (see Figure 5.18). Two-sided MWW tests on each individual task rating resulted in large p-values. Noticeable differences are the ratings of the control group on task T2, T3, T5, and T6. In all cases the control group rated the three tasks as "Very difficult" in difference to the experimental group.

5.5.4 Discussion of the Results

Overall we found evidence that the participants of the experimental group solved the tasks of our study in less time and with a higher correctness. The SQA-Mashup approach was rated as more "user friendly" than the given baseline tools reflected by a significantly higher SUS score.

When analyzing these differences on the level of the individual tasks we found major insights in the benefits of monitoring the CI-process and the capabilities of existing CI-tools. On the one hand, regarding single aspects of

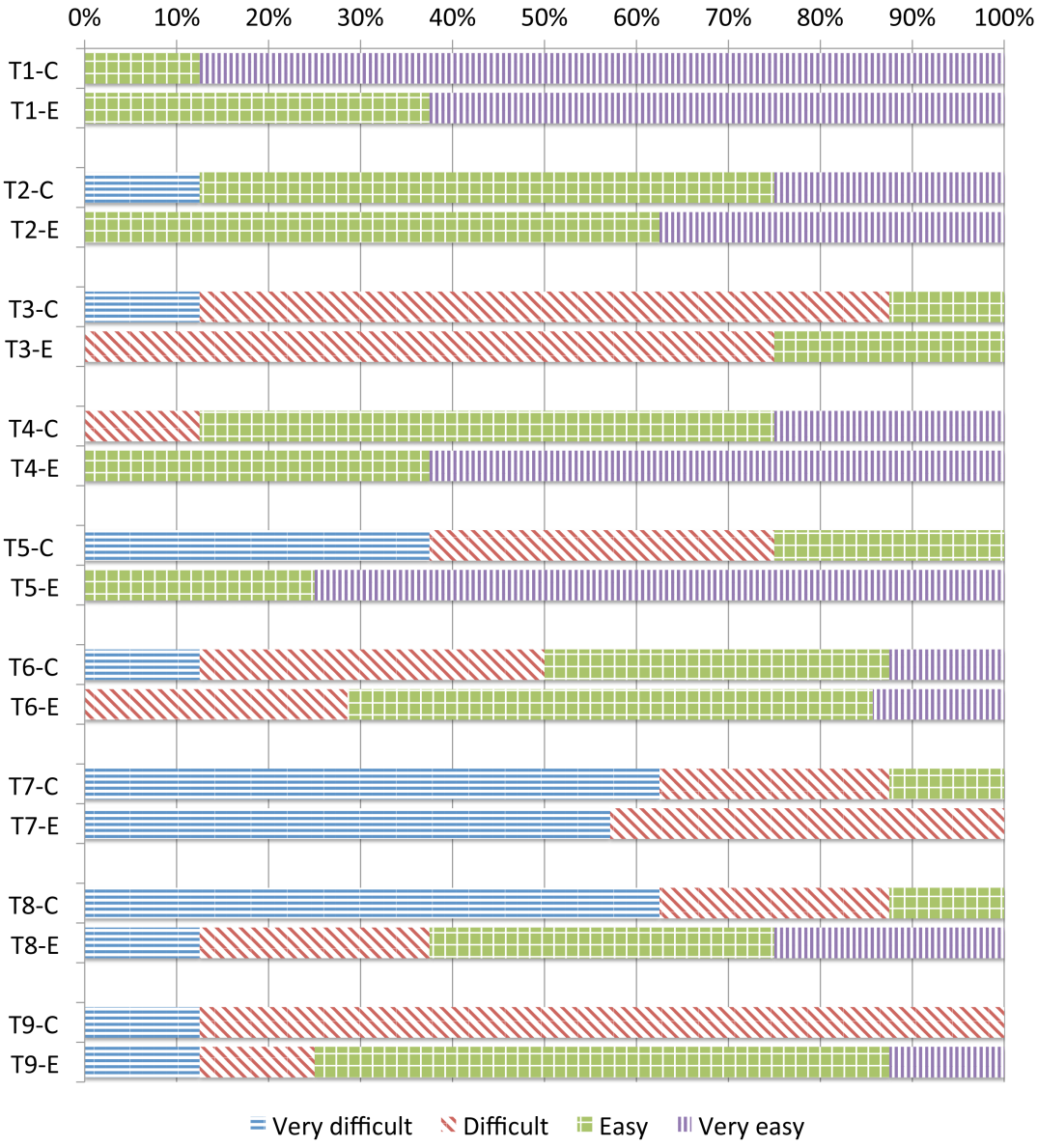

Figure 5.18: Difficulty Rating - (C)ontrol group and (E)xperimental group

software quality, such as project size (T1) or code complexity (T2), existing CI-tools provide already good support. The participants of both groups achieved a high correctness, and we could not observe a significant time gain in either of the two groups.

On the other hand, we found evidence that monitoring software quality during CI can substantially benefit from an integrated view. This is particularly the case when the required information is scattered over multiple CI-tools and displayed in different scales and units, for instance, as it is the case for tasks T3 or T8. The subjects of the control group had to search the CI-toolchain and then manually combine the information obtained from the individual tools. This seems to be a disadvantage compared to the experimental group that was presented a single interface that integrates all relevant quality information. Therefore, we see potential for such integrated approaches when multiple facts need to be merged to answer an aggregated quality aspect of a software system.

5.5.5 Threats to Validity

Unequal distribution of experience in terms of development experience and experience with the used tools can bias the results of a user study. We addressed this issue by a random assignment of the participants to a group. The analysis of both experiences with the MWW test indicated no significant difference of the two populations.

The *number of participants* might have influenced the drawn conclusion. Many results had a clear tendency but they were not statistically significant. A greater number of participants might help to draw a clear conclusion for those results.

Learning effects caused by related tasks, which have to be solved within a short period of time. The correctness and completion time results may have been affected by learning effects because every group used a fixed set of tools to solve all tasks. This threat may especially apply for the tasks T7-T9. In each of these three tasks the occurrence time of an event is important for the interlinking across the CI-tools. Only the analysis of task T8 and T9 showed that score increased in both groups and almost equally.

Incorrect task completion times might have been reported by the participants. Each participant himself was responsible for tracking and writing down the

time needed to solve a task. We monitored the time tracking activity of the participants during the study but it is still possible that incorrect task times have been reported.

Generalizability of the results. This threat is in respect to the participants of our study which were experienced but not professional developers. We tried to overcome this by selecting students with development experience in industry.

Authenticity of the tasks for the daily work of a software developer. The task selection of our user study is only a small extract of tasks which are done during software maintenance. We selected tasks from literature to avoid any favor in the direction of the baseline tools or SQA-Mashup.

5.6 Related Work

The goal of our approach is to support the answering of common questions in the context of software evolution and continuous integration. We follow insights of prior research analyzing the information needs of developers to ensure that our set of questions is relevant and useful. We divide the field of the related work into the following areas: software evolution, information needs, continuous integration, and tool integration in Software Engineering.

Software evolution: Mens et al. [Mens et al., 2005] list a number of challenges in software evolution. One approach to reason about the evolution of software systems is *the integration of data from a wide variety of sources*. The Moose project [Nierstrasz et al., 2005] is a platform, which addresses the integration of data from different of data sources, such as source code repositories or issue trackers. FM3 is the meta-model used in Moose, which builds the base for an automatic evolution analysis. Another approach based on Semantic Web technologies was proposed by Li and Zhang [Li and Zhang, 2011]. Our approach tackles the integration differently since we do not use a meta-model, such as Moose, to analyze the evolution of a software system. Instead, we integrate data with what is already computed by state-of-the-art CI-tools.

Information needs: Aranda and Venolia [Aranda and Venolia, 2009] inves-

tigated source code histories in repositories to identify common bug fixing patterns. Breu et al. [Breu et al., 2010] extracted a catalog of questions from bug reports. They found that many questions are not explicitly stated in issues and therefore often not answered. Other studies conclude with an explicit catalog of questions asked by a group of stakeholders. LaToza and Myers [LaToza and Myers, 2010] came up with a list of *Hard-to-answer questions about code*. Fritz and Murphy [Fritz et al., 2010] provide a collection of developer's questions in combination with an information fragment model to answer these questions. Another group of studies examined the tasks of professionals during software comprehension. Roehm et al. [Roehm et al., 2012] showed that developers follow different strategies to complete their work. Müller and Fritz [Müller and Fritz, 2013] shifted the focus from software testers and developers to a more diverse audience such as requirements engineers and product line managers. They found that these stakeholders require multiple different artifacts to perform their daily activities. Their findings extend existing studies [Fritz et al., 2010, LaToza and Myers, 2010] performed with software developers.

Continuous integration: Staff and Ernst reported on a controlled human experiment to evaluate the impact of continuous testing onto the source code quality [Saff and Ernst, 2004]. The results of their experiment showed an improvement caused by the immediate feedback from the continuous testing framework. Their approach was implemented as plugins for Eclipse and Emacs. A similar experiment was conducted by Hurdugaci and Zaidman [Hurdugaci and Zaidman, 2012]. They implemented a plugin for Visual Studio, which *helps developers to identify the unit tests that need to be altered and executed after a code change* [Hurdugaci and Zaidman, 2012]. Both experiments showed that an immediate feedback from CI-tools fosters software quality during software development and maintenance. We base upon these results and aim at mashing up the proper information for the respective stakeholder.

Tool integration in Software Engineering: Wasserman [Wasserman, 1990] described various types of tool integration to show their potential for computer-aided software engineering engineering (CASE) environments. The missing standards for tool integration in CASE restrict the integration of data mainly

on a tool-to-tool level. Michaud et al. [Michaud et al., 2001] proposed an approach to support the exploration of source code and documentation of Java programs. They used existing standalone tools to extracted data from source code or documentation and integrated for a visualization based on the SHriMP technique [Storey et al., 1997].

5.7 Conclusions

The establishment of a CI-toolchain can prevent a software project from descending into *Continuous Integration Hell*, but the effort to overlook the amount of data generated by CI-tools in a toolchain increases with each additional tool.

We introduced a mashup and visualization framework for CI-data called SQA-Mashup, which tailors the data according to the information needs of stakeholders, such as software developer and testers. SQA-Mashup implements a pipe-and-filter based data integration approach and a role-based visualization approach built on a dynamic view concept. We derived information needs and tasks from related works for the design and the evaluation of our proof-of-concept implementation.

We showed that the SQA-Mashup approach can (1) foster visibility of the project status, (2) improve team awareness, and (3) help spreading consistent quality information. A controlled user study with 16 participants was conducted to support this claim. In summary, the results of our controlled user study provide evidence for the following:

- An integrated view on CI-data that effectively combines multiple CI-tools can positively influence the monitoring of software quality within a software project. The participants of our study, which used SQA-Mashup achieved a 21.6% higher correctness when answering questions about the evolution and quality of a software project than those using standalone CI-tools.

- Our integrated approach leads to significant time savings in monitoring

software quality data compared to the use of standalone CI-tools. The tasks were solved 57% faster with SQA-Mashup compared to standalone CI-tools.

- The overall user satisfaction based on the System Usability Score with our integrated approach is significantly better compared to the user satisfaction with the standalone CI-tools used in the control group of our study.

Our approach serves as *one-stop shop* for software quality data monitoring in a software development project. It enables easy access to CI-data which otherwise is not integrated but scattered across multiple CI-tools. Our dynamic visualization approach allows for a tailoring of integrated CI-data according to information needs of different stakeholders such as developers or testers.

The prototypical implementation of SQA-Mashup is available in the Windows 8 App Store and a walk-through can be found on our Web-site[4].

Future work will focus on further visualizations of integrated CI-data, especially for time-related information (e.g., commit and build events visualized on a timeline). We also plan to enable a search for reoccurring patterns, for example, to find more than n commits within some minutes before a build failure.

[4] http://www.ifi.uzh.ch/seal/research/tools/sqa-mashup

6

A Framework For Rapid and Systematic Software Quality Assessment

Martin Brandtner, *Philipp Leitner, and Harald Gall*
Under submission at international journal

Abstract

Software quality assessment monitors and guides the evolution of a software system based on quality measurements. Continuous Integration (CI) environments can provide measurement data to feed such continuous assessments. However, in modern CI environments, data is scattered across multiple CI tools (e.g., build tool, version control system). Even small quality assessments can become extremely time-consuming, because each stakeholder has to seek

for the data she needs. In this paper, we introduce an approach to enable rapid and systematic software quality assessments. In our work, systematic stands for the ability to do a comprehensive quality assessment based on data from different CI tools, and rapid stands for an increase efficiency by tailoring information. Based on our findings from mining software repositories and best-practices of practitioners, we present an approach to (i) integrate CI data, (ii) profile stakeholders, (iii) tailor integrated data, and (iv) present it in accordance to the information needs of a stakeholder. We employed an empirical and a user study to evaluate the core concepts of our approach. Additionally, we carried out a case study to show the potential of our framework. The evaluation results clearly indicate that SQA-Cockpit can indeed enable more comprehensive quality assessment within less time.

6.1 Introduction

The aim of a software quality assessment (SQA) is a comprehensive description of a software system with a clear focus on quality aspects. There are various ways and aspects that can be followed or addressed to achieve this aim. According to Crosby, there is a simple reason why a regular assessment is that important for software projects: *"it is not quality that is expensive, but rather the lack of it"* [Crosby, 1979]

Ideally, SQA is accomplished during the whole life-cycle of a software system, starting with the first code file and ending with the fade out in production. Today, modern Continuous Integration (CI) environments cover almost the complete life-cycle of a system. Despite CI follows a different aim than SQA, its establishment is an important element to gather data for SQA. The data available in modern CI environments covers build related aspects (e.g., build failure), testing aspects (e.g., test failures), quality measurements (e.g., metrics), and release management aspects (e.g., commit/issue management). Those data represent the evolution of a system and an important data source for SQA.

However, a regular assessment of all the data available in CI environments

is time-consuming and depending on the size of a software system even impossible. The large number of covered aspects per integration run and the short integration cycles (multiple integration runs per day) raise the need for a mechanism to automatically filter out less relevant data. A further challenge is the scattering of the data within a CI environment. Modern CI tools provide dashboards to visualize the data generated during an integration run. In such dashboards, stakeholders of a quality assessment have to individually seek the needed information. This is an extremely time-consuming task, because each tool presents its own data only. The scattering raises the need for an integration of the data generated by different tools within a CI environment into a single user interface.

In this paper, we present an approach to enable rapid and systematic quality assessments by integrating and tailoring CI data based on the profile of stakeholders. A systematic quality assessment enables more comprehensive analysis based on integrated CI data. A tailored presentation reduced information seeking time, which enables rapid quality assessment. We derive the profiles of stakeholders from their activities in CI tools, such as version control system (VCS) or issue tracker. For example, stakeholders with a high commit and merge activity are associated to the *Integrator* profile. In contrast to traditional roles (e.g., software engineer), profiles can be used to describe the actual tasks of a stakeholder within a certain time range. This difference is especially important for a tailoring of CI data during a quality assessment. Before a tailoring is possible, the independently stored CI data has to be integrated. We present a data linking approach based on identifiers and in combination with an analysis of the timely relationship of artifact changes. An artifact in the context of our work can be, for example, a commit in the VCS or an issue in the issue tracker, or a build in the build system. For our investigations, we analyzed the evolution of in total 20 Java projects hosted by the Apache Software Foundation (ASF), JBoss, and Spring. We mined data from build tools (e.g., Jenkins-CI), quality management platforms (e.g., SonarQube), version control systems (e.g., Git), and issue tracking platforms (e.g., JIRA) within a time period of two years. In addition, we interviewed practitioners from

industry to gain a deeper understanding of best practices for source code integration and release management.

We further present a framework called *SQA-Cockpit*, which was implemented according to the findings of our work. In a case study with Apache projects, we evaluated the potential of the tool for a continuous SQA. The results of the case study clearly indicate that the benefits of the integration and profiling concepts supplement each other in approaches, such as SQA-Cockpit, and can indeed enable rapid and systematic software quality assessments.

The proposed SQA-Cockpit framework is based on previous investigations in the field of (i) best practices for continuous integration, (ii) profiling of stakeholder activities in CI tools [Brandtner et al., 2015b], and (iii) the tailoring and presentation of integrated CI data for different stakeholder groups [Brandtner et al., 2014, Brandtner et al., 2015a].

The remainder of this work is structured as follows. In Section 6.2, we introduce a motivating example to illustrate the challenges addressed in this work. Section 6.3 describes the SQA-Cockpit approach. A prototypical implementation of the SQA-Cockpit framework is presented in Section 6.4. In Section 6.5 we present the setup and the results of the case study. The most relevant related work is presented in Section 6.6. Finally, we conclude with our main findings in Section 6.7.

6.2 Motivating Example

The following example illustrates a quality assessment in an open-source project hosted by the ASF. This project is developed in Java, and uses Jenkins-CI, JIRA, GitHub, and SonarQube as CI tools. A quality assessment in this project has such stakeholders: *Shane the software engineer*, *Tim the test engineer*, and *Luci the lead engineer*. At the beginning of every assessment, the stakeholders independently inspect the evolution of the project from different perspectives. For example, Shane starts with an inspection of newly introduced coding violations. He is primarily interested in violations that have a severity of *medium*, *high*, or *blocker*. In a further step, Shane digs deeper into

selected violations. He first has a look onto the underlying commit in the VCS. In most of the cases, the violations are caused by a sub-optimal implementation chosen by the source code contributor. However, in some cases it is necessary that Shane looks up an issue description in the issue tracker to understand the purpose of a source code change. It may happen that a contributor is aware of the introduced violations, but the importance of, for example, fixing a bug was higher than the quality of the system. *Tim* on the other hand, uses a slightly different approach to assess the quality of a software system. As a test engineer, he is primarily interested in the evolution of the project's test coverage and the test cases. He starts with an inspection of issues that have been recently marked as resolved in the issue tracker. For each issue, he looks up the associated commits in the VCS to check if the contributing stakeholder has provided or updated the test code as well. Stakeholders provide, for example, fixes for an issue which is not covered by the existing test code. Many times, contributors forget to update the according test code, which prevents from an early detection of a reoccurring of the same issue. *Luci* follows yet another assessment approach. She typically starts with a look on the build status and the number of opened, reopened and resolved issues in the issue tracker. One important measure for her is the ratio of resolved to (re-)opened issues. She has to act in case of a relatively high number of opened, and especially, reopened issues compared to the number of closed issues. For example, a high number of reopened issues can be an indicator for a sub-optimal quality of the contributed source code. In such a case, Luci has to investigate the reasons. Potential reasons can be the complexity of the affected code or shortcomings of programming skills.

In the following, we illustrated the information gathering process without and with an integrated approach, such as SQA-Cockpit.

6.2.1 Scenario with state-of-the-art CI environment

In this scenario, we show how the information gathering of the described stakeholders takes place in a state-of-the-art CI environment.

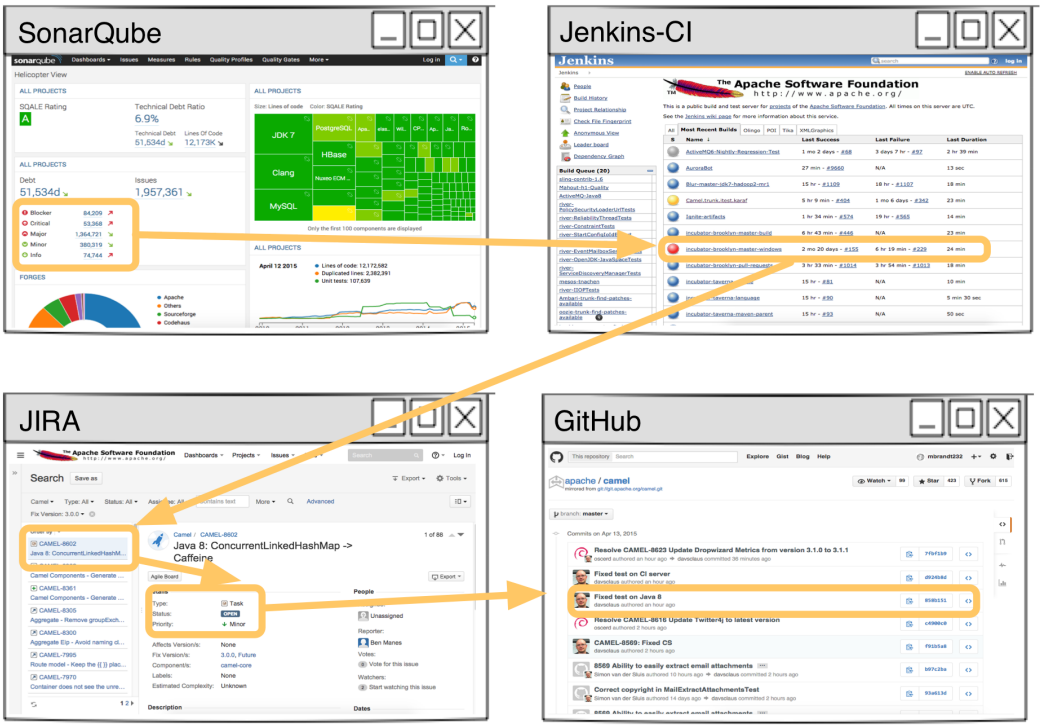

Figure 6.1: Information gathering in a modern CI environment

Luci, Shane, and Tim need to access multiple CI tools (e.g., Jenkins-CI, JIRA, GitHub, and SonarQube) in order to get the data they need for their quality assessment. Figure 6.1 depicts the dashboards provided by the different CI tools. The highlighted (orange) elements represent data that is queried by a stakeholder in certain order (orange arrows). The selected elements vary, depending on the needs of a stakeholder. For example, Shane is less interested in data from the issue tracker than Tim or Luci. In short, different stakeholders need different data. However, modern CI tools provided rather static dashboard, which cannot satisfy the information needs of all stakeholders. In such an environment, it is extremely time-consuming to seek data for a quality assessment. First of all, a user has to switch between the different tools. Second, the artifacts (e.g., issues, commits) stored in the different CI tools are not linked. For example, Shane wants to lookup the quality of a commit. In

modern CI tools, Shane has to manually seek the according commit in the quality management platform.

6.2.2 Scenario with SQA-Cockpit

In this scenario, we show how the information gathering of the described scenario takes place with an integrated approach, such as SQA-Cockpit.

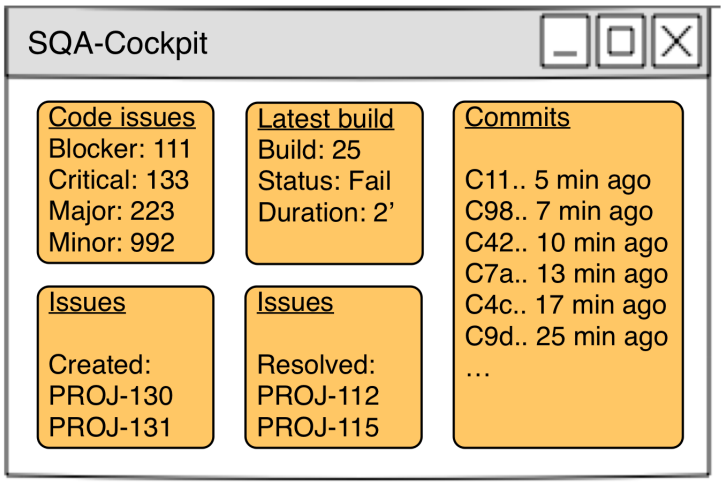

Figure 6.2: Information gathering with SQA-Cockpit

Figure 6.2 depicts the proposed dashboard provided by the SQA-Cockpit approach as a mashup. Again, the orange elements indicate data that is relevant for a certain stakeholder. The elements are connected to each other to additionally foster the information seeking. Every view is automatically composed based on the activity profile of a stakeholder. Such a profile can be derived, for example, from the event logs of CI tools. The major differences of the SQA-Cockpit approach compared to state-of-the-art CI environments are (i) the integration of the data into one screen, and (ii) a personalized view based on a stakeholders activity profile.

Luci, Shane, and Tim can be seen as representatives for different groups

of people in a software project. In modern CI tools, the presentation of data is not optimized for any of these groups. The views in these tools usually contain parts of data from various areas, which cannot satisfy all the needs of the mentioned groups. This increases the information seeking effort for each individual stakeholder every time an SQA is conducted. To overcome these limitations, we propose the SQA-Cockpit approach, which provides following mechanisms to enable an optimal workflow during an SQA:

- integration of data that is scattered across CI tools.
- tailoring of integrated CI data based on the needs of stakeholders.
- presentation of tailored data for a faster information seeking process.

6.3 Enabling Rapid and Systematic SQA

The aim of the SQA-Cockpit approach is to enable rapid and systematic SQA by integrating, tailoring, and presenting data originating from different CI environments into a uniform and seamless accessible form.

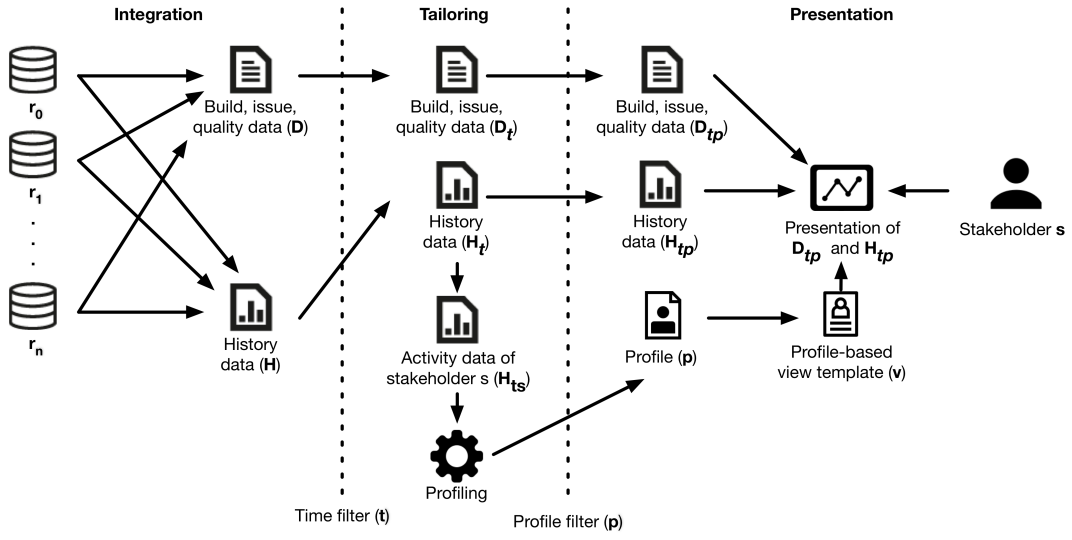

Figure 6.3: SQA-Cockpit – Overview

Figure 6.3 depicts an overview of the three phases of the SQA-Cockpit approach: *Integration, Tailoring, Presentation*. The first phase encapsulates the extraction and linking of artifact data (e.g., build, commit, issues) as well as the processing of history logs. The outcome of the first phase consists of extracted artifact and history data. In the second phase, artifact and activity data is tailored to fit into a predefined time frame and for a certain stakeholder profile. The stakeholder profile is generated based on the activity history a stakeholder. The result of this phase are artifact, activity, and profile data prepared for the presentation to a stakeholder. In the third phase, the prepared data of the previous step is presented to a stakeholder. We use *templates* to enable an automatic view composition to present the data in a meaningful way.

The used CI tools, as well as the setup of CI environments, vary depending on a software project. To address this diversity, the underlying concept of our approach does not dependent on a specific set of tools used in a CI environment. In the following, we introduce a formal model to describe the processing steps of our approach in a tool-independent and meaningful way.

6.3.1 Integration Phase

The integration phase of the SQA-Cockpit approach covers the extraction and linking of artifacts (e.g., builds, commits, issues) and history data of stakeholders. Modern CI tools, such as JIRA, provide application program interfaces (APIs) to enable the extraction of stored data (e.g., issues). However, the format and structure of the data export is not standardized and varies between different tools. Furthermore, the extraction of activity data is challenging, because modern CI tools do not provide a mechanism to explicitly track the tool usage of stakeholders.

Figure 6.4 depicts an overview of the integration phase of the SQA-Cockpit approach. The figure shows a simplified version of the data integration process based on data from the CI tools r_0 and r_1. In the following, we assume that r_0 is an issue tracker and r_1 a VCS. The extracted artifacts are issues in case of r_0 and commits in case of r_1. To foster the understanding of our approach, we

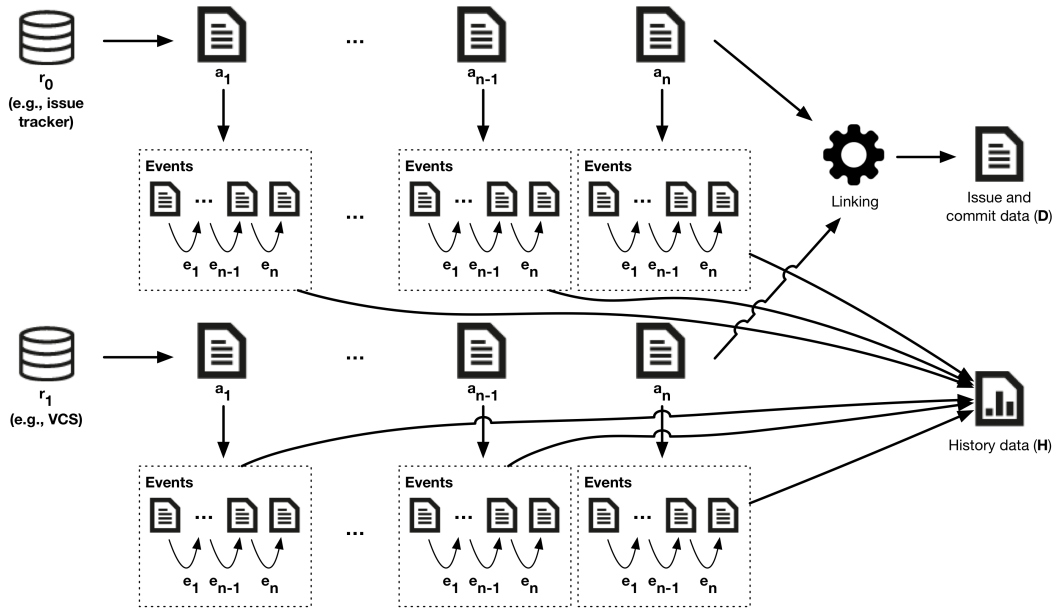

Figure 6.4: SQA-Cockpit – Artifact and Activity Data Integration

further assume that the data format of the extracted artifacts is the same for all CI tools. One example for such a data format can be the JavaScript Object Notation (JSON), which is supported by modern CI tools.

We define the following tuple for all artifacts (A) extracted from CI tools:

$$A =< id, data, hist, date > \tag{6.1}$$

The *id* element represents a unique identifier of an artifact. For example, a 40-bit SHA1 key in case of a Git commit or an issue key in case of an issue stored in JIRA. The actual data of an artifact extracted from a CI tool is represented by the *data* element, which we assume to be plain text in JSON notation. In addition to the latest data, the histories *hist* of all artifacts is extracted. The values of the *hist* element are a subset of H associated with an artifact, which is represented in our model by following condition: $hist \subseteq H$. The element *date* contains the latest modification date of an artifact. The tuple of a history H is defined as follows:

$$H =< date, stakh, change > \tag{6.2}$$

The *date* element represents a timestamp of a change event in a history. In addition, the user name of the performing stakeholder is represented by the *stakh* element. The actual change is represented in the *change* element. A change contains the name of the changed element (e.g., issue status) as well as the old (e.g., open) and the new value (e.g., closed) of it.

The next step of the integration phase is the linking of artifacts A from different CI tools. For example, a commit that resolves an issue and triggers a build is linked to both, the issue and the build. However, such links are not explicitly stored in artifacts extracted from CI tools. We define the following tuple D to represent the linking:

$$D =< id, data, hist, date, links > \tag{6.3}$$

$$\forall x \in A \rightarrow x \in D \tag{6.4}$$

The tuple D is derived from A, and the data of A is mapped to D. Therefore, the meaning of the element *id*, *data*, *hist*, and *date* is the same as in A. The *links* element is new and contains references to related artifacts. As mentioned before, modern CI tools do not explicitly store references between artifacts. A simple approach to reconstruct such references can be the use of the time to describe the relationship of two artifacts. For example, an issue is marked as resolved immediately after the commit of the source code change. Our investigations [Schermann et al., 2015] showed that in most cases developers indeed close an issue immediately after the according commit. However, there is no guarantee that a developer follows this procedure every time. We propose a linking approach that combines to use of artifact identifiers and the timely relationship of events. We define our linking approach as follows:

$$\begin{aligned} &\forall x \in A : links(x) := \\ &\{y \in A : x \neq y \wedge contains(y.data, x.id) \wedge |x.date - y.date| < 5min\} \end{aligned} \tag{6.5}$$

The first condition $x \neq y$ ensures that an artifact is not linked to itself. The function *contains(y.data, x.id)* in the second condition ensures the existence of a unique identifier in the data of a related artifact. It is defined as follows:

$$contains(stack, needle) := \{\exists string \in stack : string = needle\} \tag{6.6}$$

For example, artifact *x* is an issue and artifact *y* is a commit. The *contains* function returns true in case that *y.data* contains the unique identifier *y.id*. The third condition addresses the timely relationship of two events. Previous research [Schermann et al., 2015] has shown that a maximum time difference of 5 minutes between two events of different artifacts lead to an optimal performance for the linking.

6.3.2 Tailoring Phase

The tailoring phase of the SQA-Cockpit approach covers the filtering and preparation of integrated data for the presentation to a certain stakeholder.

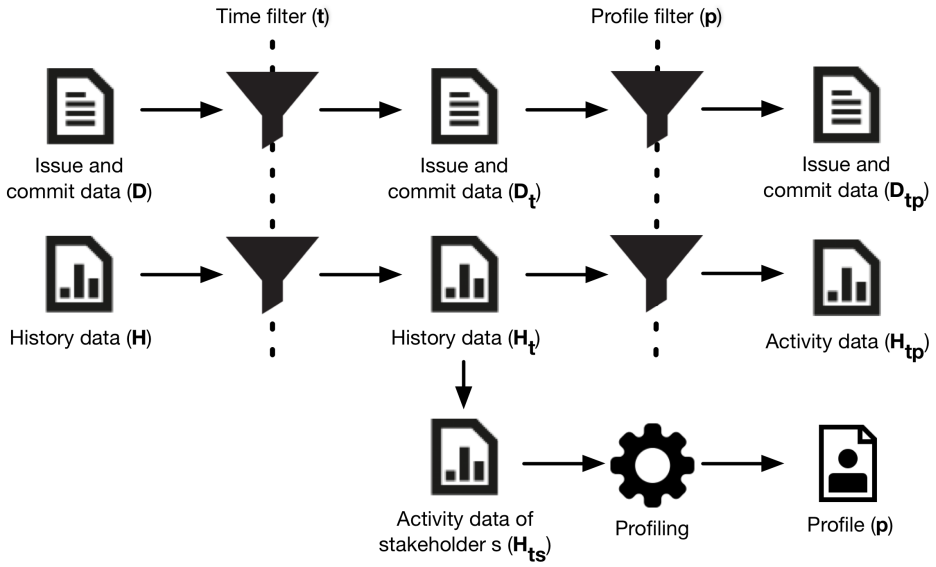

Figure 6.5: SQA-Cockpit – Data Tailoring and Stakeholder Profiling

Figure 6.5 depicts an overview of the tailoring phase. This figure shows the filtering of artifact data D and history data H, as well as the profiling of a stakeholder based on her activity data. We continue the example case consisting of an issue tracker (r_0) and a VCS (r_1), which was introduce in Section 6.3.1.

Filtering. The SQA-Cockpit approach follows a two-phased filtering for artifact data D and history data H. First, all artifacts and activities are filtered based on the time frame t, which is used for a quality assessment. An example for an often used time frame is 14 days, according to our interview partners from industry. We define the time filter for artifact and history data as follows:

$$D_t = \{x \in D \mid \exists y \in x.hist(y.date \geq (t_{now} - t))\} \tag{6.7}$$

$$H_t = \{x \in H \mid x.date \geq (t_{now} - t)\} \tag{6.8}$$

In case of D, elements are filtered if they do not have at least one history entry in the selected time frame. Elements of H are filtered in case there is no entry within the time frame. The results of the first filtering are artifacts D_t and histories events H_t within a time frame t.

Second, all artifact and history data are filtered based the profile of a stakeholder. We define a stakeholder S as follows:

$$S =< stakh, profile, activity > \tag{6.9}$$

The *stakh* element contains the user name of a stakeholder. This element is the same as the *stakh* element of H. The second element *profile* contains the profile of a stakeholder. Possible values of this elements are a subset of P: $profile \subseteq P$. The *activity* element contains history entries that have been caused by the activity of a stakeholder in CI tools. We define the activity element as follows:

$$\forall x \in S : activity(x) := \{y \in H_t \mid x.stakh = y.stakh\} \tag{6.10}$$

A profile is defined as follows:

$$P = \{id, metrics\} \tag{6.11}$$

Every profile P consists of a unique identifier *id* and a set of metric names. Each *metrics* element describes metrics in which a stakeholder is interested if she has this profile. A metric can be, for example, a quality metric or the number of closed issues.

We use the SQA-Profiles approach [Brandtner et al., 2015b] to derive the profile of a stakeholder. In the following, we introduce a method *sqaProfile(activity)*, which represents the computation of an SQA-Profile. The mapping of a SQA-Profile to a profiles P is defined as follows:

$$\forall x \in S : profile(x) := \{\exists p \in P : p.id = sqaProfile(x.activity)\} \tag{6.12}$$

Based on the profile definition, the filtering of artifacts and history data can be defined as follows:

$$D_tP = \{x \in D_t \mid \exists y \in P(containsM(x.data, y.measurement)\} \tag{6.13}$$

$$H_tP = \{x \in H_t \mid \exists y \in P(containsH(x.log, y.measurement)\} \tag{6.14}$$

The *containsM* and *containsH* functions check if a certain metric is part of an artifact or a history. We define these functions as follows:

$$containsM(stack, needle) := \{\exists aEntry \in stack : aEntry = needle\} \tag{6.15}$$

$$containsH(stack, needle) := \{\exists hEntry \in stack : hEntry = needle\} \tag{6.16}$$

It is important to mention that the history data H_{tp} will be presented to a

stakeholder, similar to the artifact data. This data contains, for example, the recent activities in a software project.

6.3.3 Presentation Phase

The presentation phase of the SQA-Cockpit approach covers the visualization of the integrated and tailored data. In the previous phase, the stakeholder profile was generated and the integrated data was tailored in accordance to the metrics described in the profile. During the presentation phase, the tailored data is visualized and arranged to address the needs of a certain profile and stakeholder.

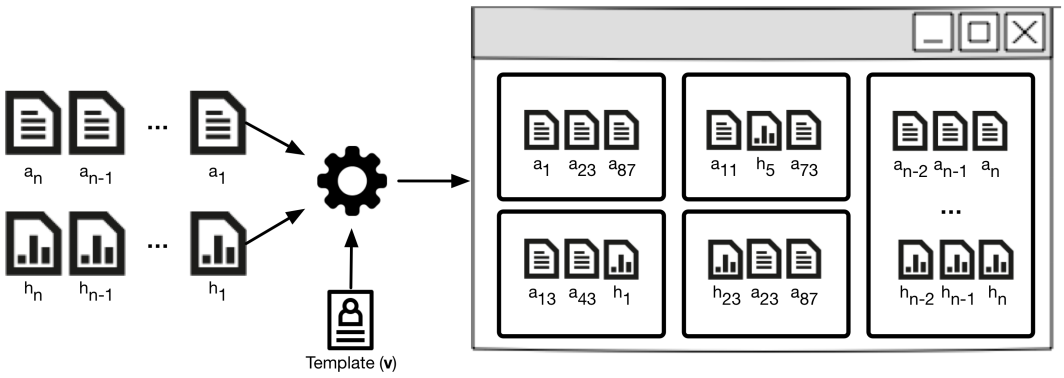

Figure 6.6: SQA-Cockpit – Presentation of Tailored Artifact and History Data

Figure 6.6 depicts an overview of the view composition process in SQA-Cockpit. The tailored artifact a_i and history h_j data are arranged in a view based on the arrangement provided by template v. It is important to mention that SQA-Cockpit presents artifact and history data in a combined way. For example, SQA-Cockpit presents artifact and related history data in the same widget of a view instead of having dedicated widgets for the different data.

The automatic view composition is based on a template concept. A tem-

plate in the context of our work is defined as follows:

$$V =< profile, metric, arrangement, widget > \tag{6.17}$$

The *profile* element describes a profile that is covered by a template. This element is defined as subset of P: $profile \subseteq P$. The *metric* element, describes the kind of data that is addressed by a template. For example, a quality metric or issue data. The *arrangement* element is used to prioritize a metric during the interface arrangement process. For example, a metric with a high prioritization gets a more prominent place in the dashboard than others. The *widget* element contains information about how a certain metric should be presented in the user interface. An example for such a widget can be a list, a bar chart, a pie chart, a tree map, etc. The view composition process takes care of the linking in the data and presents artifacts and histories grouped by their relationships. In a next step, the data entities get visualized as widgets based on the parameterization in the template.

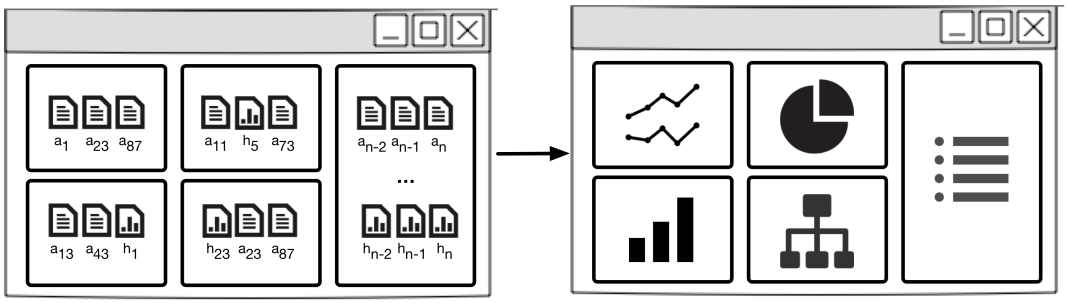

Figure 6.7: SQA-Cockpit – Mapping of Data to Widgets for Presentation

Figure 6.7 depicts a transition of integrated data to a widget-based user interface. We propose well established visualization approaches, such as charts, lists, or tree maps, because of their familiarity to stakeholders of a software project.

6.4 SQA-Cockpit - A Framework for Rapid and Systematic SQA

We developed a framework called *SQA-Cockpit*, which is based on the presented three phased approach with the same name. The three phases are reflected in the architecture of SQA-Cockpit. Figure 6.8 depicts an overview of the layered architecture used for the design of the prototypical framework.

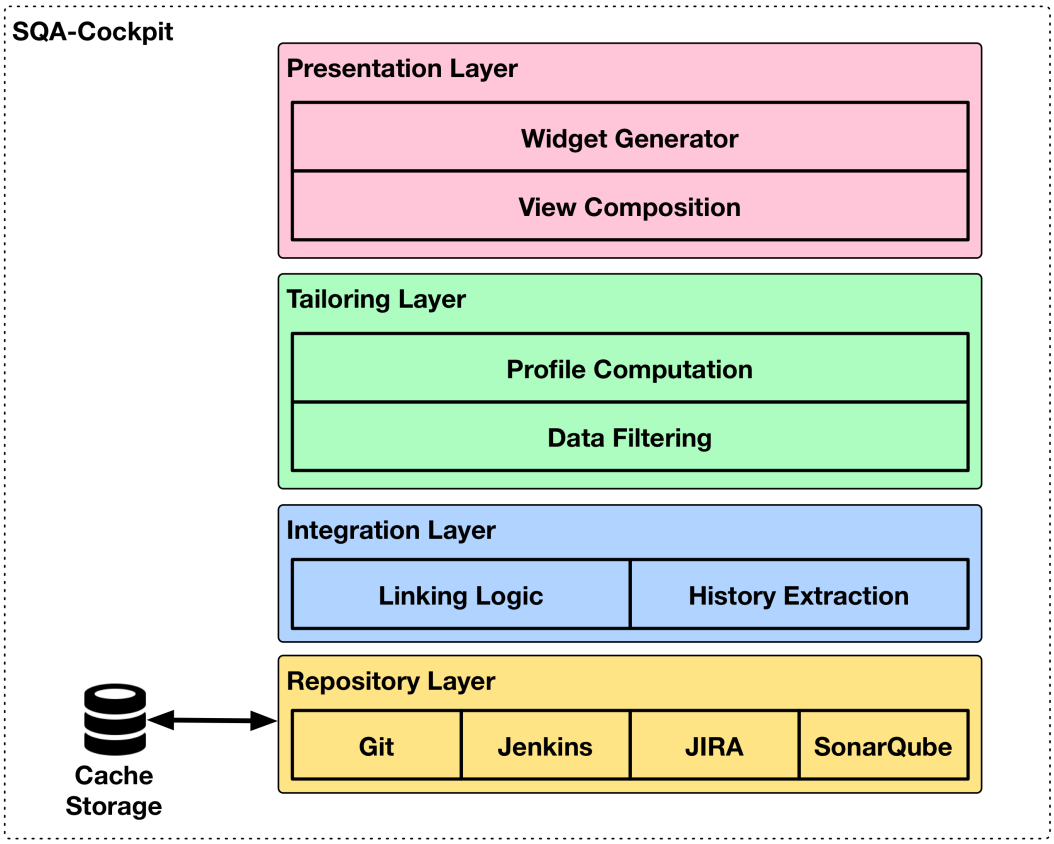

Figure 6.8: SQA-Cockpit – Framework Architecture

The first layer in the architecture is the *Repository Layer*. This layer takes care of the framework's communication with APIs provided by CI tools. Within

this layer, a plug-in based concept is followed to cover the different APIs of CI tools. We use a cache storage to ensure the availability of data and the querying performance for the further processing steps. The data fetching and caching takes place transparent for all further layers. The second layer is called *Integration Layer*. This layer addresses the linking of artifact data and the extraction of history data. *Tailoring Layer* is the name of the third layer. In this layer, the data gets filtered for a predefined time frame and profile. The output of the filtering is used as input for the generation of the stakeholder profiles and for the data presentation in the next layer. The last layer of the SQA-Cockpit is called *Presentation Layer*. Within this layer, the integrated and tailored data is prepared for the presentation in the user interface. During the view composition, the artifact and history data are arranged based on the priorities provided by templates. The final processing step in this layer is the generation of the widgets for the visualization of the data in the user interface. Figure 6.9 depicts the layout of a view generated by the SQA-Cockpit framework. This view shows build, commit, quality and issue data of the Apache PDFBox project within a week.

The visual appearance of views in SQA-Cockpit is similar for all profiles, but the views differ in the presented data. For example, the treemap depicted in Figure 6.9 on the right top can contain data for different profiles and time spans. It can show source code changes of a developer within a day or within a week. For example, a software engineer might prefer a treemap that contains data of a few days only, because she has better understanding of her source code changes during the last day. A leading engineer might prefer a longer time span, because she is not aware of each source code change and wants an overview of hotspots in the changed artifacts.

Our prototypical implementation of the SQA-Cockpit framework is available for download as Google Chrome Application[1]. The SQA-Cockpit framework is implemented in JavaScript. It uses the D3.js[2] library for the visualization and the clusterfck[3] library for the clustering.

[1] http://www.ifi.uzh.ch/seal/people/brandtner/projects/sqa-cockpit
[2] http://d3js.org/
[3] https://github.com/harthur/clusterfck

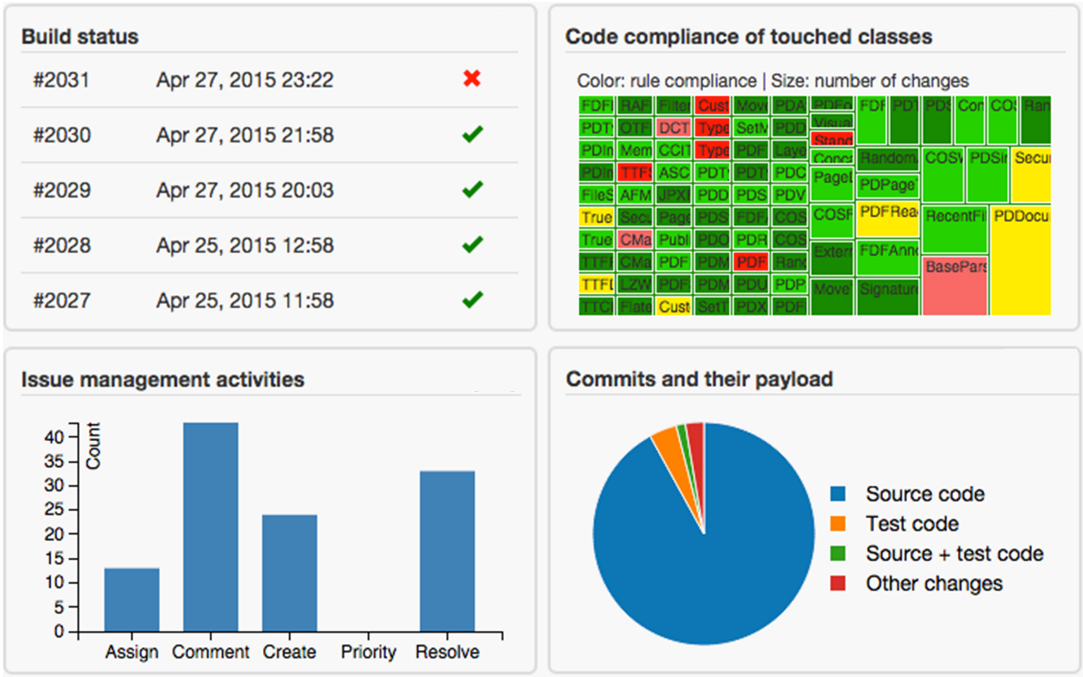

Figure 6.9: SQA-Cockpit – Standard View Layout

6.5 Evaluation

We performed a case study to show the potential of SQA-Cockpit for a comprehensive quality analysis based on data integrated from different CI tools. The result of this study is a comparison of modern CI tools and SQA-Cockpit based on their capabilities for a systematic software quality assessment.

We further evaluated the capabilities of SQA-Cockpit to enable rapid software quality assessment. In a first step, we evaluated the performance of the stakeholder profiling, which is required for the information tailoring. In a second step, we evaluated the impact of tailored information onto the efficiency of a software quality assessment.

We decided not to present an evaluation of the used data integration concepts, because these concepts have been evaluated in previous research (e.g., [Schermann et al., 2015, Nguyen et al., 2012, Wu et al., 2011, Fischer et al., 2003a]).

6.5.1 Case Study

We surveyed the literature for questions that are asked during the initial step of a software quality assessment. For example: *"What have people been working on?"* [Fritz et al., 2010] or *"Who is working on what?"* [Fritz et al., 2010]. We decided to use a generalized form of these question for our case study: *"What recent changes have been made?"* [LaToza and Myers, 2010]. In the following, we describe possible ways to answer this questions with modern CI tools and with the SQA-Cockpit framework. For this case study, we used data from CI tools of the Apache PDFBox project.

Figure 6.10 depicts screenshots of recent changes listed in Jenkins-CI (i), GitHub (ii), JIRA (iii), and SonarQube (iv).

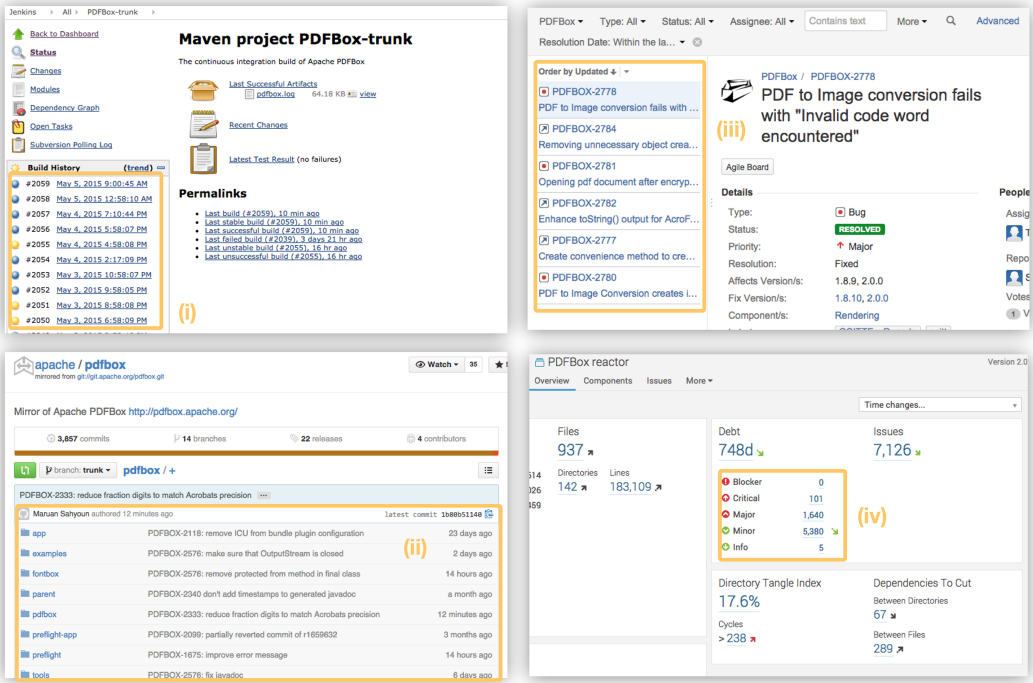

Figure 6.10: Dashboards of modern CI tools

Jenkins-CI lists the latest builds with their status (build or failed), a build

number, and a timestamp. A click onto a list entry presents more information (e.g., changed resources) of the selected build. A stakeholder can use the list provided by Jenkins-CI to easily lookup the latest builds and potential causes of a build break. Jenkins-CI provides an optional linking between builds and commit ids.

GitHub provides a view with the latest commits, which can be useful to answer the question about recent changes. This list contains for each commit information about the committer, the time, and the changed resources. While it is possible to get information about single commits, a systematic analysis of the commits is almost impossible. Following question is an example, which would require a systematic analysis: *Which class has been changed most?* [Fritz et al., 2010] The data to answer this question is in the VCS, but modern CI tools do not offer capabilities for a systematic analysis.

JIRA does not provide a view or list of recent changes in the default setup. However, the JQL query language provided by JIRA enables the composition of views for such a purpose. Queries can be easily composed via the user interface of JIRA. Every query can be stored as view and used for quality assessments. In addition, JIRA provides an optional linking of commits to issues. Based on the issue key provided in a commit message, JIRA integrates the data of the according commits in the issue tracker.

SonarQube follows a different approach to present recent changes of the generated quality metrics. The values displayed in the dashboard represent the latest measurements and a small indicator next to each measurement represents the direction of the recent change. This indicator-based approach can only be used to visualize the evolution of the last two values of a metric, which is a limitation compared to change logs of other tools.

Figure 6.11 depicts a default dashboard provided by the SQA-Cockpit framework and a set of detail views, which allow for a deeper analysis.

SQA-Cockpit provides various widgets for the presentation of recent changes in a software system. For example, a list of the last builds, a treemap of changed classes, a bar chart with issue changes, and a bar chart with coding violations.

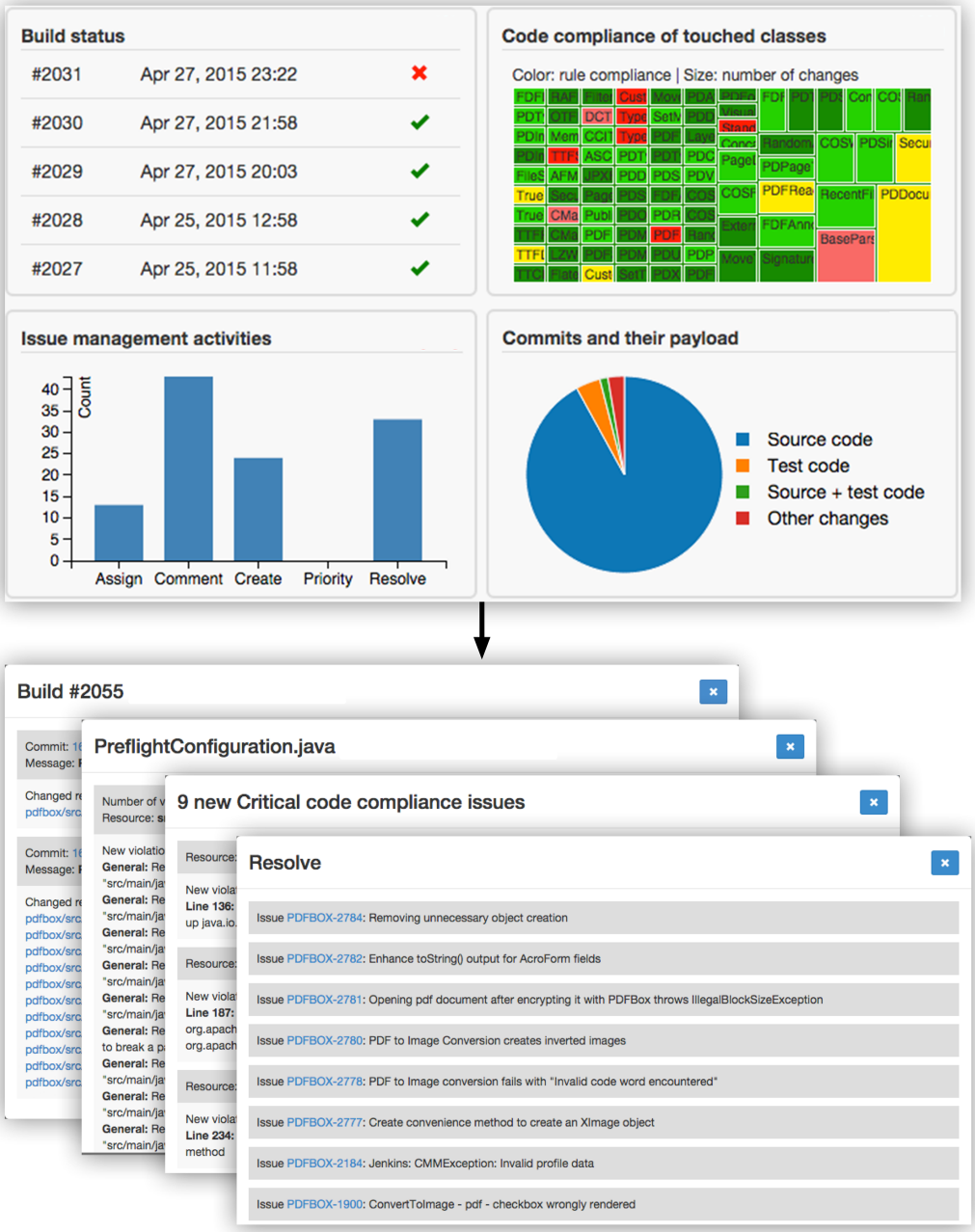

Figure 6.11: Dashboard and Detail Views of SQA-Cockpit

A list of builds presented in SQA-Cockpit has the same format as the according list in Jenkins-CI. It is possible to click on an entry to get more detailed information of a build. The detail view presents a list of the commits and changed resources, which were part of a build. For more information, links are provided to the commits in the VCS and to issue entries in the issue tracker. A major difference Jenkins-CI and SQA-Cockpit is the stronger linking to other artifacts, such as commits and issues.

A treemap of changed classes in SQA-Cockpit can answer multiple questions at once. This treemap presents only classes that have been changed recently, for example, within the last 14 days. The size of an element in the treemap indicated the number of code changes that have been applied onto a class. At the same time the color indicates the evolution of the coding violations. For example, a big element that is colored dark red indicates a class that has been changed multiple times while the number of coding violations has increased. A click on a treemap element opens a detail view with the coding violations and links to the associated commits in the VCS. This widget especially addresses a questions, which is hard to answer with modern CI tools: *Which class has been changed most?* [Fritz et al., 2010]. Furthermore, it is easier to detect newly introduced coding violations compared to a manual seeking in the quality management platform.

A bar chart with issue changes provides an overview of issue management activities. Modern CI tools, such as JIRA, always present a snapshot of the issue management. For example, 55 issues in status open with a major priority. SQA-Cockpit presents statistics about the issue changes within a certain time span. For example, 24 comments on issues or 5 priority changes. In a detail view, a list of all issues presents the according changes and links to the item in the issue tracker. This way of presenting issue management activities is beneficial compared to the way of modern CI tools in case of questions about recent changes in the issue tracker.

A bar chart with coding violations in SQA-Cockpit complements the treemap to detect bad coding styles. While the treemap widget is focused on the correlation of code change sizes and their impact on coding violations,

the bar chart provides an overview of coding violations only. In case of a normal distribution of changes and introduced violations a treemap can lead to misleading results. This widget is similar to the visualization that is used in SonarQube. However, the detail view of SQA-Cockpit shows only newly introduce violations and provides a seamless linking to the underlying commits in the VCS.

	Jenkins-CI	GitHub	JIRA	SonarQube	SQA-Cockpit
Links to artifacts in other CI tools	P	N	P	N	Y
Composition of individual views	N	N	P	Y	Y
Filtering based on the activity of a stakeholder	N	N	Y	N	Y
Query language	N	N	Y	N	N
Time-based filtering	N	N	Y	Y	Y
Views with data from multiple CI Tools	N	N	N	N	Y

Table 6.1: Feature Support per CI Tool

Table 6.5.1 lists the differences that we found in our case study. *Y* indicates that the according feature is supported, *N* indicates that the according feature is not supported, and *P* indicates that the according feature is partially supported. The partial linking support in case of Jenkins-CI and JIRA is caused by the fact that both tools support a linking to no other system than the VCS. The view composition in JIRA is rated as partially supported because it heavily relies on the query engine, which is explicitly listed in the Table 6.5.1. The case study

showed that SQA-Cockpit outperforms existing CI tools in their linking and filtering capabilities.

6.5.2 Stakeholder Profiling

The automatic profiling of stakeholders enables the profile-based data filtering of SQA-Cockpit. During the tailoring phase, activity data extracted from a VCS and an issue tracker are examined to derive stakeholder profiles. These profiles are required for the tailoring and the presentation of the integrated data. In this evaluation, we inspect the precision and recall of an automatic and ruled-based profiling of stakeholders compared to a semi-automatic and machine learning-based ones [Brandtner et al., 2015b].

Setup. We extracted activity data of Project Management Committee (PMC) members from the VCS and the issue tracker of 20 Apache projects. More precisely, we used the number of commits and merges in the VCS as well as the number of events related to issue assignee, comment, priority, and status changes to describe the activities of a stakeholder. We decided to use activity data of PMC members only, because they have an explicit role in Apache projects. Furthermore, a list of PMC members for the different Apache projects is available on a website of the Apache Software Foundation. We used the machine learning-based profiling approach to compute a set of profiles and to derive ruled-based profile descriptions [Brandtner et al., 2015b]. In a next step, we extracted the activity data of all stakeholders that have contributed to any of the 20 Apache projects. The rule-based profile descriptions as well as the extracted activity data of all stakeholders were used as input data for the profile computation with the SQA-Cockpit framework.

Results. We compared the set of stakeholders associated to profiles generated by SQA-Cockpit with the results of the machine learning-based approach. Table 6.5.2 lists the precision, recall and F-measure achieved by our rule-based approach compared to a baseline based on machine learning.

A *true-positive* (TP) is any stakeholder-profile association that is in accor-

Profile	TP	FP	Total	Precision	Recall	F-measure
P1	3	1	3	0.75	1.00	0.86
P2	9	1	9	0.90	1.00	0.95
P3	9	5	12	0.64	0.75	0.69
P4	80	2	106	0.98	0.75	0.85
Overall	*101*	*9*	*130*	*0.92*	*0.78*	*0.84*

Table 6.2: Rule-based classification - Performance

dance with the classification of the baseline dataset and a *false-positive* (FP) is any stakeholder-profile association that is not part of the baseline dataset.

In total, our approach classified 101 stakeholders correctly (*true-positive*), 9 stakeholders to a wrong profile (*false-positive*), and 20 stakeholders kept unclassified (*false-negatives*). This leads to an overall precision of 0.92, a recall of 0.78, and a F-measure of 0.84 compared to the baseline. The results showed that the rule-based and project-independent SQA-Profiles works and can be used to automatically profile stakeholders based on their activity data.

6.5.3 Presentation

The dynamic composition of views enables the profile-based data presentation of SQA-Cockpit. Profiles based on activity data can be used for a tailoring of large amounts of integrated quality data. The presentation of tailored data is as important as the tailoring it self to enable rapid and systematic software quality assessments. To estimate the impact of an integrated view on quality data on information seeking, we formulated two hypotheses (see Table 6.5.3) and verified them through statistical tests based on the results of a user study [Brandtner et al., 2015a].

Setup. We ran a user study with 16 participants. Each subject had at least five years of development experience in general and minimum three years of development experience with Java. The youngest participant was 22 years and the oldest 49 years with a median age of 27.9 years. We separated the

ID	Null Hypothesis
$H1_0$	There is no difference in the total score per subjects between the experimental and the control group.
$H2_0$	There is no difference in the total time per subjects between the experimental and the control group.

Table 6.3: Hypotheses

participants into a control and experimental group. Both groups consisted of eight randomly assigned participants each. We decided to use Jenkins-CI, GitHub, and SonarQube as baseline CI tools in the control group. The participants of the experimental group had to use a prototypical version of our integrated front-end [Brandtner et al., 2015a]. The data for both groups were gathered from the same instances of CI tools. Each participant of our study had to solve in total nine tasks. The tasks address program comprehension, testing, and cross-domain topics. We selected the JUnit project [Louridas, 2005], which is a popular unit testing framework for source code written in Java.

The maximum time to work through all tasks of our study was 45 minutes, with a maximum of five minutes per task. After five minutes, a participant had to stop working and to continue with the next question. One study session lasted in total 40-60 minutes, as there was no time restriction on the introduction and the feedback section.

Results. The maximum total score a study participant could achieve was 27 points. The achieved median of the total scores were 20.0 points in the control group and 25.5 points in the experimental group. In other words, this median difference of 5.5 points means that the experimental group outperformed the control group on average by 20.4% (=5.5/27) when referring to the total score per subject.

The median of the total time was 30.7 minutes in the control group and 17.5 minutes in the experimental group. This median difference of 13.2 minutes means that the experimental group outperformed the control group on average by 29.3% (=13.2/45) when referring to the total time per subject.

The box-plot of the total score per subject (see Figure 6.12, left) of the experimental group shows that the 0.75 quantile and the median are close and almost the same as the upper whisker of the plot. This indicates a skewed distribution. Furthermore, Q-Q plots and Shapiro-Wilk tests showed significant evidence against normality in the data. We made a similar observation regarding the total time (see Figure 6.12, right). Since our group sample sizes are below the commonly accepted rule of thumb (at minimum 30 to 40) we chose a non-parametric, independent-two-samples procedure, i.e., Mann-Whitney-Wilcoxon (MWW), to test all hypotheses.

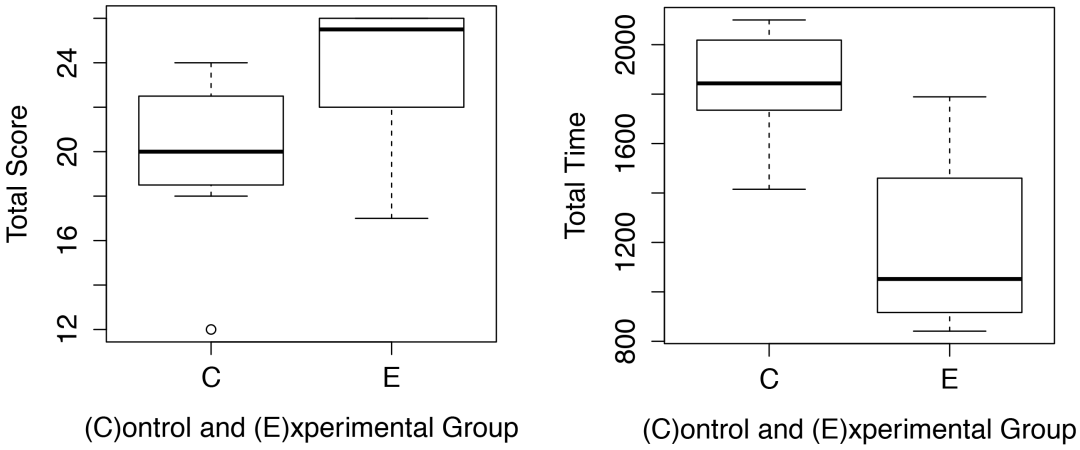

Figure 6.12: Evaluation - Total Score/Time

Under the above considerations, $H1_0$ (see Table 6.5.3) states that the mean ranks of total score per subject of the two groups are equal. A two-sided MWW test resulted in a p-value of 0.038 and 0.95 (non-parametric) confidence interval bounds of [1, 7]. This gives us significant evidence against $H1_0$. The total scores per subject in the experimental group are significantly different than for the control group. Based on these results, we find significant evidence that the subjects of the experimental group score higher.

$H2_0$ (see Table 6.5.3) states that the mean ranks of total time per subject of the two groups are equal. A two-sided MWW test resulted in a p-value of

0.003 and (non-parametric) 0.95 confidence interval bounds of [-977, -308]. This gives us significant evidence against $H2_0$: The total time per subjects in the experimental group is significantly different than for the control group. Based on these results we find strong evidence that the subjects of the experimental group could solve the tasks in less time.

The results showed that the integrated view concept works and reduces information seeking time and foster a correct interpretation.

6.6 Related Work

The goal of our approach is to enable rapid and systematic software quality assessments. For this, we rely on related work from various research areas. The most relevant research areas that are related to our work are: software evolution, artifact linking, continuous integration, and information needs.

Software evolution. Mens et al. [Mens et al., 2005] list a number of challenges in software evolution. One approach to reason about the evolution of software systems is *the integration of data from a wide variety of sources*. The Moose project [Nierstrasz et al., 2005] is a platform, which addresses the integration of data from different of data sources, such as source code repositories or issue trackers. FM3 is the meta-model used in Moose, which builds the base for an automatic evolution analysis. Another approach based on Semantic Web technologies was proposed by Li and Zhang [Li and Zhang, 2011]. Our approach tackles the integration differently since we do not use a meta-model, such as Moose, to analyze the evolution of a software system. Instead, we integrate data with what is already generated by modern CI-tools.

Artifact Linking. Ayari et al. [Ayari et al., 2007] investigated threats on building models for commit and issue data. Their results showed that a considerable number of links cannot be established based on a numerical issue id in the commit message. Yet another approach was introduced by Bachmann and Bernstein [Bachmann and Bernstein, 2009]. They proposed a set of software engineering process measurements based on commit and issue data. A survey of five open source projects and one closed source project showed major differ-

ences in the engineering process expressed by the commit and issue data. In further work [Bachmann et al., 2010], Bachmann et al. established a ground truth of commit and issue data from the Apache Httpd project created by a core developer of the project. Their analysis showed that bugs get often fixed without an issue entry and that commits not always change functionality of a software system. They introduced a tool call Linkster [Bird et al., 2010], which provides capabilities for manual commit and issue linking. Our artifact linking approach can be seen as a combination that cherry picks elements of previous research approaches.

Continuous integration. Staff and Ernst reported on a controlled human experiment to evaluate the impact of continuous testing onto the source code quality [Saff and Ernst, 2004]. The results of their experiment showed an improvement caused by the immediate feedback from the continuous testing framework. Their approach was implemented as plugins for Eclipse and Emacs. A similar experiment was conducted by Hurdugaci and Zaidman [Hurdugaci and Zaidman, 2012]. They implemented a plugin for Visual Studio, which *helps developers to identify the unit tests that need to be altered and executed after a code change* [Hurdugaci and Zaidman, 2012]. Both experiments showed that an immediate feedback from CI-tools fosters software quality during software development and maintenance. We base upon these results and aim at integrating the proper information for the respective stakeholder.

Information needs. Aranda and Venolia [Aranda and Venolia, 2009] investigated source code histories in repositories to identify common bug fixing patterns. Breu et al. [Breu et al., 2010] extracted a catalog of questions from bug reports. They found that many questions are not explicitly stated in issues and therefore often not answered. Other studies conclude with an explicit catalog of questions asked by a group of stakeholders. LaToza and Myers [LaToza and Myers, 2010] came up with a list of *Hard-to-answer questions about code*. Fritz and Murphy [Fritz et al., 2010] provide a collection of developer's questions in combination with an information fragment model to answer these questions. Another group of studies examined the tasks of professionals during software comprehension. Roehm et al. [Roehm et al., 2012] showed that developers

follow different strategies to complete their work. Müller and Fritz [Müller and Fritz, 2013] shifted the focus from software testers and developers to a more diverse audience such as requirements engineers and product line managers. They found that these stakeholders require multiple different artifacts to perform their daily activities. Their findings extend existing studies [LaToza and Myers, 2010, Fritz et al., 2010] performed with software developers. Our approach is focused especially on information needs during a software quality assessment.

To best of our knowledge, there no work that combines all the mentioned research areas to enable rapid and systematic software quality assessments.

6.7 Conclusion

Today's software quality assessments suffer from the scattering of information across different dashboards and tools. In this paper, we presented our SQA-Cockpit framework to help overcome the information scattering by enabling rapid and systematic software quality assessment: *systematic* to perform a comprehensive analysis of a software system based on data integrated from different CI tools; and *rapid* to increase the efficiency during an assessment by tailoring integrated CI data to the information needs of a stakeholder.

The dynamic CI data integration provided by SQA-Cockpit enables more comprehensive quality analysis of a software system within less time compared to a use of standalone CI tools. Further, a tailoring of integrated CI data based on stakeholder profiles leads to significant efficiency gains by avoiding long lasting information seeking steps. In an evaluation, the biggest gains were achieved in cases where a stakeholder had to utilize data from multiple CI tools that use different time spans or scales for data presentation. A potential drawback of the automatic tailoring provided by SQA-Cockpit is the missing possibility to query individual data elements from the bulk of integrated CI data.

Overall, the findings of our work can be summarized as follows: (1) integration of CI data enables more comprehensive software quality assessment;

(2) profiles derived from activity data are a powerful mechanism to describe information needs of a stakeholder; (3) tailoring of integrated CI data based on profiles increases the efficiency within a software quality assessment; and (4) the SQA-Cockpit framework enables a flexible integration and tailoring of integrated CI data for rapid and systematic software quality assessment.

The presented approach can ignite a new direction of research as well as tooling, which enables individualized dashboards for stakeholders based on their needs. The presented stakeholder profiles provide an initial set of activity profiles that exist in Apache software projects. Further investigations are needed to extract and describe stakeholder profiles that exist in other open source software projects as well as in industrial software projects.

Bibliography

[An et al., 2014] An, L., Khomh, F., and Adams, B. (2014). Supplementary bug fixes vs. re-opened bugs. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 205–214.

[Antoniol et al., 2008] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., and Guéhéneuc, Y.-G. (2008). Is it a bug or an enhancement? In *Proceedings of the 2008 conference of the center for advanced studies on collaborative research meeting of minds*, page 304.

[Antoniol et al., 2000] Antoniol, G., Canfora, G., Casazza, G., and De Lucia, A. (2000). Information retrieval models for recovering traceability links between code and documentation. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 40–49.

[Anvik et al., 2006] Anvik, J., Hiew, L., and Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 361–370, New York, NY, USA. ACM.

[Anvik and Murphy, 2007] Anvik, J. and Murphy, G. C. (2007). Determining Implementation Expertise from Bug Reports. In *Fourth International Workshop on Mining Software Repositories*, page 2.

[Apache Software Foundation,] Apache Software Foundation. How should i apply patches from a contributor?

[Aranda and Venolia, 2009] Aranda, J. and Venolia, G. (2009). The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 298–308, Washington, DC, USA. IEEE Computer Society.

[Ayari et al., 2007] Ayari, K., Meshkinfam, P., Antoniol, G., and Di Penta, M. (2007). Threats on building models from cvs and bugzilla repositories: The mozilla case study. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 215–228, Riverton, NJ, USA. IBM Corp.

[B. Kitchenham and S. Pfleeger, 2008] B. Kitchenham and S. Pfleeger (2008). Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*. Springer London.

[Bacchelli et al., 2009] Bacchelli, A., D'Ambros, M., Lanza, M., and Robbes, R. (2009). Benchmarking lightweight techniques to link e-mails and source code. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 205–214, Washington, DC, USA. IEEE Computer Society.

[Bacchelli et al., 2010] Bacchelli, A., Lanza, M., and Robbes, R. (2010). Linking e-mails and source code artifacts. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 375–384, New York, NY, USA. ACM.

[Bachmann and Bernstein, 2009] Bachmann, A. and Bernstein, A. (2009). Software process data quality and characteristics: A historical view on open and closed source projects. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, pages 119–128, New York, NY, USA. ACM.

[Bachmann et al., 2010] Bachmann, A., Bird, C., Rahman, F., Devanbu, P., and Bernstein, A. (2010). The missing links: Bugs and bug-fix commits. In

Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pages 97–106, New York, NY, USA. ACM.

[Bettenburg et al., 2008] Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. (2008). Extracting structural information from bug reports. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 27–30, New York, NY, USA. ACM.

[Bird et al., 2009] Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., and Devanbu, P. (2009). Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 121–130, New York, NY, USA. ACM.

[Bird et al., 2010] Bird, C., Bachmann, A., Rahman, F., and Bernstein, A. (2010). Linkster: Enabling efficient manual inspection and annotation of mined data. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 369–370, New York, NY, USA. ACM.

[Bird et al., 2006] Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 137–143, New York, NY, USA. ACM.

[Bird et al., 2008] Bird, C., Pattison, D., D'Souza, R., Filkov, V., and Devanbu, P. (2008). Latent social structure in open source projects. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 24–35.

[Bird and Zimmermann, 2012] Bird, C. and Zimmermann, T. (2012). Assessing the value of branches with what-if analysis. In *Proceedings of the ACM*

SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 45:1–45:11, New York, NY, USA. ACM.

[Bissyande et al., 2013] Bissyande, T., Thung, F., Wang, S., Lo, D., Jiang, L., and Reveillere, L. (2013). Empirical evaluation of bug linking. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 89–98.

[Brandtner et al., 2014] Brandtner, M., Giger, E., and Gall, H. (2014). Supporting continuous integration by mashing-up software quality information. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 184–193.

[Brandtner et al., 2015a] Brandtner, M., Giger, E., and Gall, H. (2015a). Sqa-mashup: A mashup framework for continuous integration. *Information and Software Technology*, 65(0):97 – 113.

[Brandtner et al., 2015b] Brandtner, M., Müller, S. C., Leitner, P., and Gall, H. C. (2015b). Sqa-profiles: Rule-based activity profiles for continuous integration environments. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER '16, pages 301–310.

[Breu et al., 2010] Breu, S., Premraj, R., Sillito, J., and Zimmermann, T. (2010). Information needs in bug reports: improving cooperation between developers and users. In *Conference on Computer-Supported Cooperative Work and Social Computing*, CSCW, pages 301–310, Savannah.

[Brooke, 1996] Brooke, J. (1996). Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189:194.

[Brooks, 1995] Brooks, Jr., F. P. (1995). *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Brun et al., 2011] Brun, Y., Holmes, R., Ernst, M. D., and Notkin, D. (2011). Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of*

Software Engineering, ESEC/FSE '11, pages 168–178, New York, NY, USA. ACM.

[Buse and Zimmermann, 2012] Buse, R. P. L. and Zimmermann, T. (2012). Information needs for software development analytics. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 987–996, Piscataway, NJ, USA. IEEE Press.

[Cheng et al., 2003] Cheng, L.-T., de Souza, C. R., Hupfer, S., Patterson, J., and Ross, S. (2003). Building Collaboration into IDEs. *Queue*, 1(9):40.

[Corley et al., 2011] Corley, C. S., Kraft, N. A., Etzkorn, L. H., and Lukins, S. K. (2011). Recovering traceability links between source code and fixed bugs via patch analysis. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '11, pages 31–37, New York, NY, USA. ACM.

[Crosby, 1979] Crosby, P. B. (1979). *Quality is free: the art of making quality certain*. New York: McGraw-Hill.

[Dabbish et al., 2012] Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). Social coding in GitHub. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, page 1277.

[Duvall et al., 2007] Duvall, P. M., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley.

[Fischer et al., 2003a] Fischer, M., Pinzger, M., and Gall, H. (2003a). Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering*, WCRE '03, pages 90–101, Washington, DC, USA. IEEE Computer Society.

[Fischer et al., 2003b] Fischer, M., Pinzger, M., and Gall, H. (2003b). Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, pages 23–32.

[Fowler, 2014] Fowler, M. (Last accessed: 09/01/2014). Continuous integration, http://www.martinfowler.com/articles/continuousintegration.html.

[Fritz et al., 2010] Fritz, T., Ou, J., Murphy, G. C., and Murphy-Hill, E. (2010). A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 385–394.

[Giger et al., 2010] Giger, E., Pinzger, M., and Gall, H. (2010). Predicting the fix time of bugs. In *Proceedings of the 2Nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA. ACM.

[Gousios et al., 2014] Gousios, G., Pinzger, M., and Deursen, A. v. (2014). An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 345–355, New York, NY, USA. ACM.

[Guo et al., 2010] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2010). Characterizing and predicting which bugs get fixed. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 1, pages 495–504.

[Guo et al., 2011] Guo, P. J., Zimmermann, T., Nagappan, N., and Murphy, B. (2011). "not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 395–404, New York, NY, USA. ACM.

[Hall et al., 2009] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software. *ACM SIGKDD Explorations Newsletter*, 11(1):10.

[Holm, 1979] Holm, S. (1979). A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70.

[Hurdugaci and Zaidman, 2012] Hurdugaci, V. and Zaidman, A. (2012). Aiding software developers to maintain developer tests. In *European Conference on Software Maintenance and Reengineering*, CSMR, pages 11–20, Szeged.

[J. Anvik and G.C. Murphy, 2011] J. Anvik and G.C. Murphy (2011). Reducing the effort of bug report triage. *ACM Transactions on Software Engineering and Methodology*, 20(3):1–35.

[Jiang et al., 2013] Jiang, Y., Adams, B., and German, D. M. (2013). Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 101–110, Piscataway, NJ, USA. IEEE Press.

[Kersten and Murphy, 2005] Kersten, M. and Murphy, G. C. (2005). Mylar. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168.

[Kim et al., 2006] Kim, S., Zimmermann, T., Pan, K., and Whitehead, E. J. J. (2006). Automatic identification of bug-introducing changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 81–90, Washington, DC, USA. IEEE Computer Society.

[Kim et al., 2007] Kim, S., Zimmermann, T., Whitehead Jr., E. J., and Zeller, A. (2007). Predicting faults from cached history. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 489–498, Washington, DC, USA. IEEE Computer Society.

[Ko and Chilana, 2010] Ko, A. J. and Chilana, P. K. (2010). How power users help and hinder open bug reporting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1665–1674, New York, NY, USA. ACM.

[Lanza et al., 2005] Lanza, M., Marinescu, R., and Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[LaToza and Myers, 2010] LaToza, T. D. and Myers, B. A. (2010). Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU, pages 8:1–8:6, Reno.

[Leah Findlater, Joanna Mcgrenere, 2008] Leah Findlater, Joanna Mcgrenere, D. M. (2008). Evaluation of a Role-Based Approach for Customizing a Complex Development Environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1267–1270.

[Li and Zhang, 2011] Li, Y.-F. and Zhang, H. (2011). Integrating software engineering data using semantic web technologies. In *Working Conference on Mining Software Repositories*, MSR, pages 211–214, Waikiki.

[Louridas, 2005] Louridas, P. (2005). Junit: unit testing and coiling in tandem. *Software, IEEE*, 22(4):12–15.

[Mann and Whitney, 1947] Mann, H. and Whitney, D. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, 18(1):50–60.

[McConnell, 2004] McConnell, S. (2004). *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

[Meneely et al., 2010] Meneely, A., Corcoran, M., and Williams, L. (2010). Improving developer activity metrics with issue tracking annotations. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, pages 75–80.

[Mens et al., 2005] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in software evolution. In *International Workshop on Principles of Software Evolution*, IWPSE, pages 13–22, Lisbon.

[Michaud et al., 2001] Michaud, J., Storey, M., and Muller, H. (2001). Integrating information sources for visualizing java programs. ICSM, pages 250–258.

[Mockus et al., 2002] Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346.

[Mockus and Herbsleb, 2002] Mockus, A. and Herbsleb, J. D. (2002). Expertise browser: a quantitative approach to identifying expertise. In *International Conference on Software Engineering*, ICSE, pages 503–512, Orlando.

[Mockus et al., 2009] Mockus, A., Nagappan, N., and Dinh-Trong, T. T. (2009). Test coverage and post-verification defects: A multiple case study. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 291–301, Washington.

[Mockus and Votta, 2000] Mockus, A. and Votta, L. (2000). Identifying reasons for software changes using historic databases. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 120–130.

[Müller and Fritz, 2013] Müller, S. C. and Fritz, T. (2013). Stakeholders' information needs for artifacts and their dependencies in a real world context. In *International Conference on Software Maintenance*, ICSM, pages 290–299, Eindhoven.

[Nguyen et al., 2012] Nguyen, A. T., Nguyen, T. T., Nguyen, H. A., and Nguyen, T. N. (2012). Multi-layered approach for recovering links between bug reports and fixes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 63:1–63:11, New York, NY, USA. ACM.

[Nierstrasz et al., 2005] Nierstrasz, O., Ducasse, S., and Gĭrba, T. (2005). The story of moose: an agile reengineering environment. *SIGSOFT Softw. Eng. Notes*, 30(5):1–10.

[Ostrand et al., 2005] Ostrand, T., Weyuker, E., and Bell, R. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355.

[Pham et al., 2013] Pham, R., Singer, L., Liskin, O., Figueira Filho, F., and Schneider, K. (2013). Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 112–121, Piscataway, NJ, USA. IEEE Press.

[Pinzger et al., 2008] Pinzger, M., Nagappan, N., and Murphy, B. (2008). Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12.

[R Development Core Team, 2011] R Development Core Team (2011). *R: A Language and Environment for Statistical Computing*. Vienna.

[Rastkar et al., 2010] Rastkar, S., Murphy, G. C., and Murray, G. (2010). Summarizing software artifacts: A case study of bug reports. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 505–514, New York, NY, USA. ACM.

[Robillard et al., 2014] Robillard, M. P., Maalej, W., Walker, R. J., and Zimmermann, T., editors (2014). *Recommendation Systems in Software Engineering*. Springer Berlin Heidelberg.

[Roehm et al., 2012] Roehm, T., Tiarks, R., Koschke, R., and Maalej, W. (2012). How do professional developers comprehend software? In *International Conference on Software Engineering*, ICSE, pages 255–265, Piscataway.

[Romo et al., 2014] Romo, B. A., Capiluppi, A., and Hall, T. (2014). Filling the gaps of development logs and bug issue data. In *Proceedings of The International Symposium on Open Collaboration*, OpenSym '14, pages 8:1–8:4, New York, NY, USA. ACM.

[Saff and Ernst, 2004] Saff, D. and Ernst, M. D. (2004). An experimental evaluation of continuous testing during development. *SIGSOFT Softw. Eng. Notes*, 29(4):76–85.

[Schermann et al., 2015] Schermann, G., Brandtner, M., Panichella, S., Leitner, P., and Gall, H. (2015). Discovering loners and phantoms in commit and issue data. In *23rd IEEE International Conference on Program Comprehension, Florence, Italy*, pages 4–14.

[Shihab et al., 2012] Shihab, E., Bird, C., and Zimmermann, T. (2012). The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 301–310, New York, NY, USA. ACM.

[Singer et al., 1997] Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. (1997). An examination of software engineering work practices. In *Centre for Advanced Studies Conference*, CASCON, pages 21–36, Toronto.

[Śliwerski et al., 2005] Śliwerski, J., Zimmermann, T., and Zeller, A. (2005). When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, pages 1–5, New York, NY, USA. ACM.

[Storey et al., 1997] Storey, M. A. D., Wong, K., Fracchia, F., and Muller, H. (1997). On integrating visualization techniques for effective software exploration. InfoVis, pages 38–45.

[Surian et al., 2011] Surian, D., Liu, N., Lo, D., Tong, H., Lim, E.-P., and Faloutsos, C. (2011). Recommending People in Developers' Collaboration Network. In *2011 18th Working Conference on Reverse Engineering*, pages 379–388.

[Surian et al., 2010] Surian, D., Lo, D., and Lim, E.-P. (2010). Mining Collaboration Patterns from a Large Developer Network. In *2010 17th Working Conference on Reverse Engineering*, pages 269–273.

[Taneja et al., 2010] Taneja, K., Zhang, Y., and Xie, T. (2010). MODA: Automated test generation for database applications via mock objects. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 289–292, New York, NY, USA. ACM.

[Tsay et al., 2014] Tsay, J., Dabbish, L., and Herbsleb, J. (2014). Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366, New York, NY, USA. ACM.

[Wasserman, 1990] Wasserman, A. (1990). Tool integration in software engineering environments. In *Software Engineering Environments*, volume 467 of *Lecture Notes in Computer Science*, pages 137–149.

[Wettel et al., 2011] Wettel, R., Lanza, M., and Robbes, R. (2011). Software systems as cities: a controlled experiment. In *International Conference on Software Engineering*, ICSE, pages 551–560, Waikiki.

[Weyuker et al., 2007] Weyuker, E. J., Ostrand, T. J., and Bell, R. M. (2007). Using Developer Information as a Factor for Fault Prediction. In *Third International Workshop on Predictor Models in Software Engineering*, page 8.

[Williams and Kessler, 2002] Williams, L. and Kessler, R. (2002). *Pair Programming Illuminated*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Wu et al., 2011] Wu, R., Zhang, H., Kim, S., and Cheung, S.-C. (2011). Relink: Recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 15–25, New York, NY, USA. ACM.

[Zimmermann et al., 2012] Zimmermann, T., Nagappan, N., Guo, P. J., and Murphy, B. (2012). Characterizing and predicting which bugs get reopened. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1074–1083.

[Zimmermann et al., 2007] Zimmermann, T., Premraj, R., and Zeller, A. (2007). Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA. IEEE Computer Society.